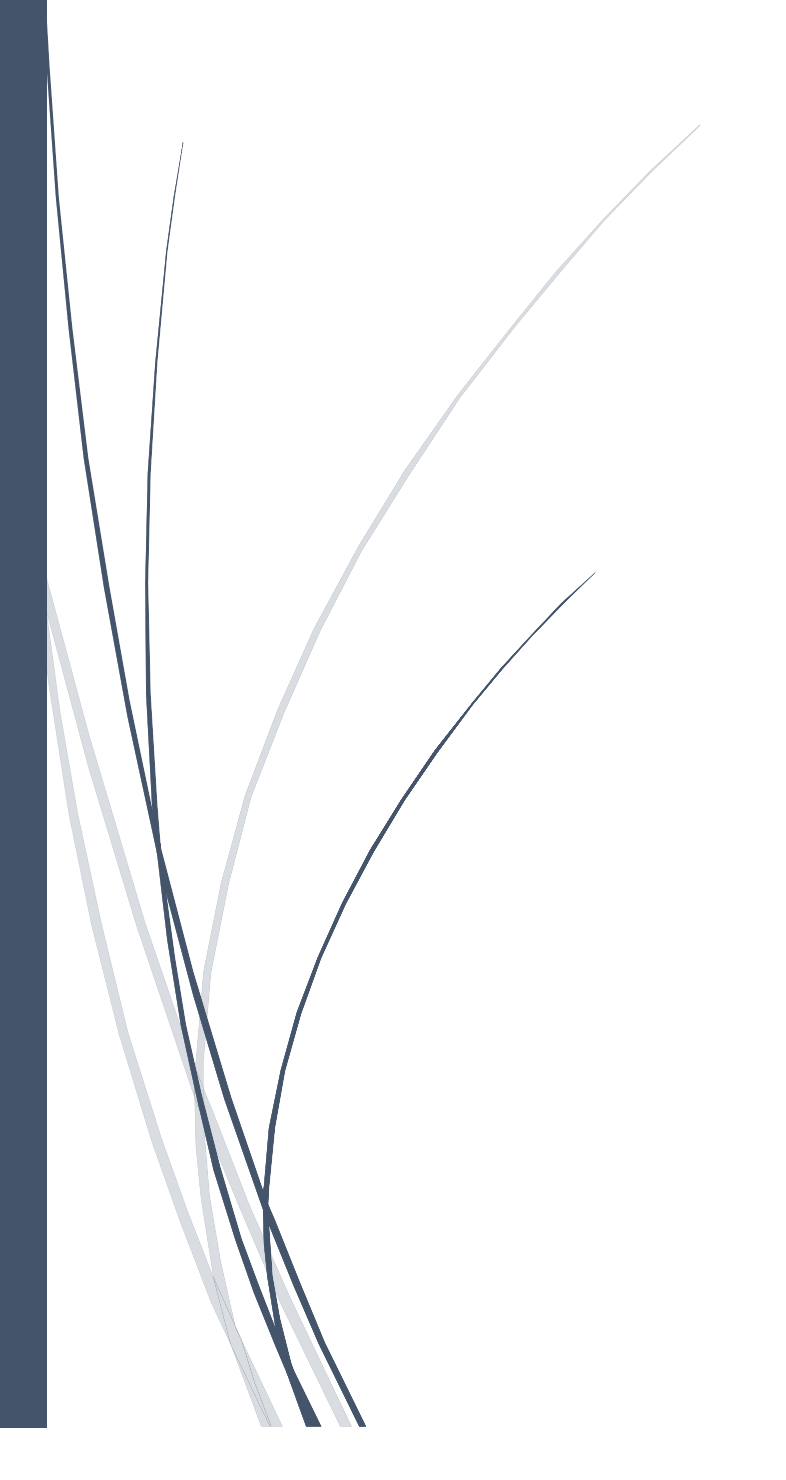
2016

Redes Neuronales

Parte 1.



Rafael Alberto Moreno Parra

Contenido

Otros libros del autor

Libro: "Segunda parte de uso de algoritmos genéticos para la búsqueda de patrones". En Colombia 2014. En publicación por la Universidad Libre – Cali.

Libro: "Un uso de algoritmos genéticos para la búsqueda de patrones". En Colombia 2013. En publicación por la Universidad Libre – Cali.

Libro: "Desarrollo fácil y paso a paso de aplicaciones para Android usando MIT App Inventor". En: Colombia 2013. Págs. 104. Ubicado en: https://openlibra.com/es/book/desarrollo-de-aplicaciones-para-android-usando-mit-app-inventor-2

Libro: "Desarrollo de un evaluador de expresiones algebraicas. **Versión 2.0**. C++, C#, Visual Basic .NET, Java, PHP, JavaScript y Object Pascal (Delphi)". En: Colombia 2013. Págs. 308. Ubicado en: https://openlibra.com/es/book/evaluador-de-expresiones-algebraicas-ii

Libro: "Desarrollo de un evaluador de expresiones algebraicas. C++, C#, Visual Basic .NET, Java, PHP, JavaScript y Object Pascal (Delphi)". En: Colombia 2012. Págs. 308. Ubicado en: https://openlibra.com/es/book/evaluador-de-expresiones-algebraicas

Libro: "Simulación: Conceptos y Programación" En: Colombia 2012. Págs. 81. Ubicado en: https://openlibra.com/es/book/simulacion-conceptos-y-programacion

Libro: "Desarrollo de videojuegos en 2D con Java y Microsoft XNA". En: Colombia 2011. Págs. 260. Ubicado en: https://openlibra.com/es/book/desarrollo-de-juegos-en-2d-usando-java-y-microsoft-xna . ISBN: 978-958-8630-45-8

Libro: "Desarrollo de gráficos para PC, Web y dispositivos móviles" En: Colombia 2009. ed.: Artes Gráficas Del Valle Editores Impresores Ltda. ISBN: 978-958-8308-95-1 v. 1 págs. 317

Artículo: "Programación Genética: La regresión simbólica".
Entramado ISSN: 1900-3803 ed.: Universidad Libre Seccional Cali
v.3 fasc.1 p.76 - 85, 2007

Página Web del autor

Investigación sobre Inteligencia Artificial: http://darwin.50webs.com
Correo: ramsoftware@gmail.com

Canal de Youtube

Canal en Youtube: http://www.youtube.com/user/RafaelMorenoP (dedicado a desarrollo de aplicaciones web en JavaScript y PHP, uso de aplicaciones Web escritas en PHP, desarrollo en C# y Visual Basic .NET)

Licencia de este libro

Licencia del software

Todo el software desarrollado aquí tiene licencia LGPL "Lesser General Public License"



Marcas registradas

En este libro se hace uso de las siguientes tecnologías registradas:

Microsoft ® Windows ® Enlace: http://windows.microsoft.com/en-US/windows/home

Microsoft ® Visual Studio 2015 ® Enlace: http://windows.microsoft.com/en-US/windows/home

Introducción

En el momento en que escribo esta primera parte del libro sobre redes neuronales, en los medios de comunicación se ha avivado un gran interés por la inteligencia artificial y uno de los temas más citados son las redes neuronales. El reconocimiento de imágenes, los automóviles autónomos o jugar partidas del juego Go venciendo a oponentes humanos expertos son noticia en la actualidad.

He trabajado en el campo de la inteligencia artificial en el tema de algoritmos genéticos y especialmente en la regresión simbólica que es dar con la mejor función matemática que explique el comportamiento de una serie de datos. Ese interés me atrajo a las redes neuronales porque una aplicabilidad de estas, es precisamente encontrar un patrón en una serie de datos.

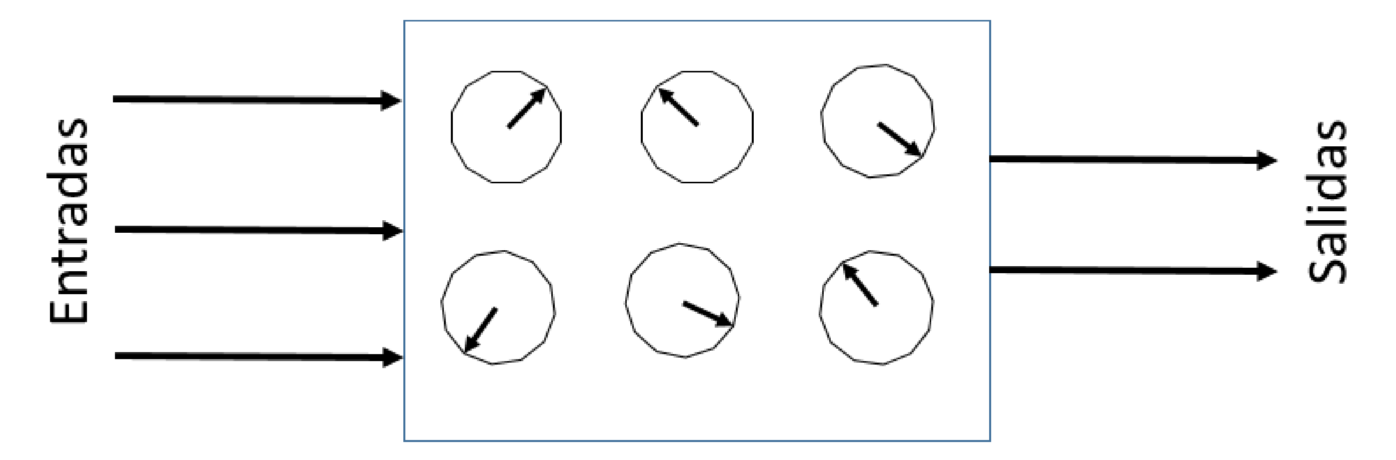
Este libro es un inicio en este fascinante campo de las redes neuronales, desde el "hola mundo" que es entrenar una red (de una sola neurona) para que aprenda la tabla del OR y del AND, luego el perceptrón multicapa (capas de neuronas interconectadas) para aprender cosas más difíciles como la tabla del XOR, reconocimiento básico de caracteres y encontrar el patrón en una serie de datos usando el algoritmo de propagación hacia atrás conocido como "backpropagation". Se explica en detalle cómo se llegan a las fórmulas usadas por ese algoritmo.

El conocimiento técnico requerido para entender los temas de este libro son algoritmos (variables, si condicional, ciclos, procedimientos, funciones), POO (programación orientada a objetos) y Visual C# (se hará uso del IDE de Microsoft Visual Studio 2015). Adicionalmente hay que estar familiarizado con una serie de conocimientos matemáticos necesarios como factorización, derivación, integración, buscar máximos y mínimos, y números aleatorios.

Aunque el código en C# se encuentra en este documento, se le facilita al lector descargarlo en http://darwin.50webs.com/Espanol/Capit07.htm , que es el código de los últimos tres grandes ejemplos.

Iniciando

Las redes neuronales son como los algoritmos: una caja negra en la cual hay una serie de entradas, la caja y una serie de salidas.

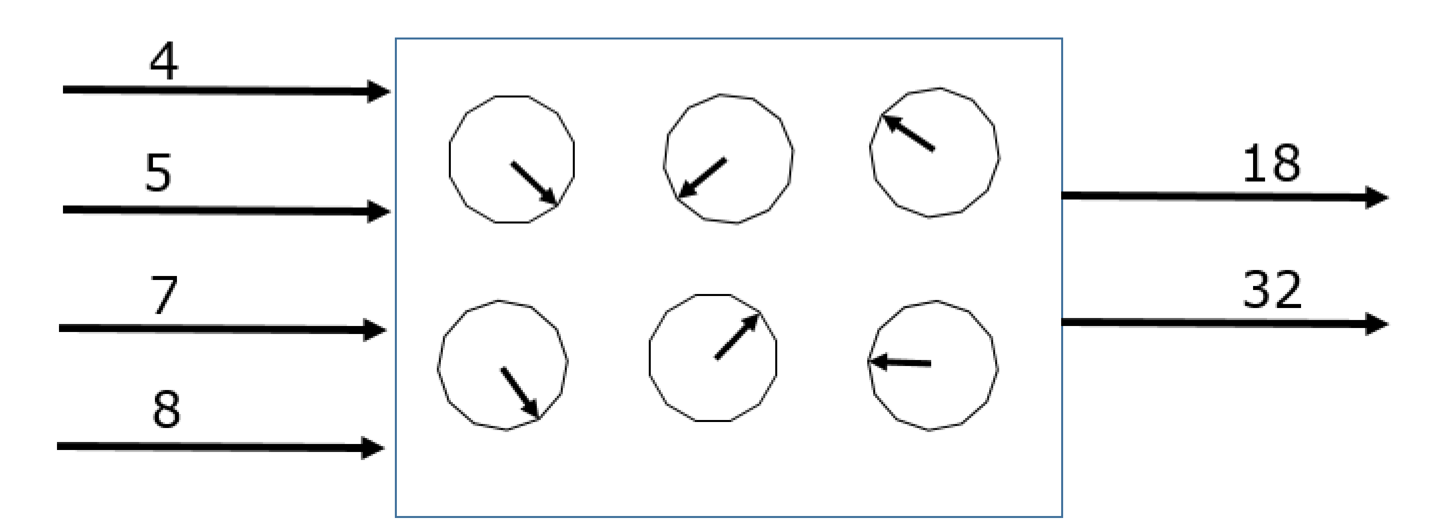


Pero hay algo especial en esa caja que representa las redes neuronales: una serie de controles analógicos, algunos girando a la izquierda y otros girando a la derecha de tal modo que su giro afecta las salidas.

En el ejemplo, supongamos que tenemos las siguientes entradas y salidas deseadas

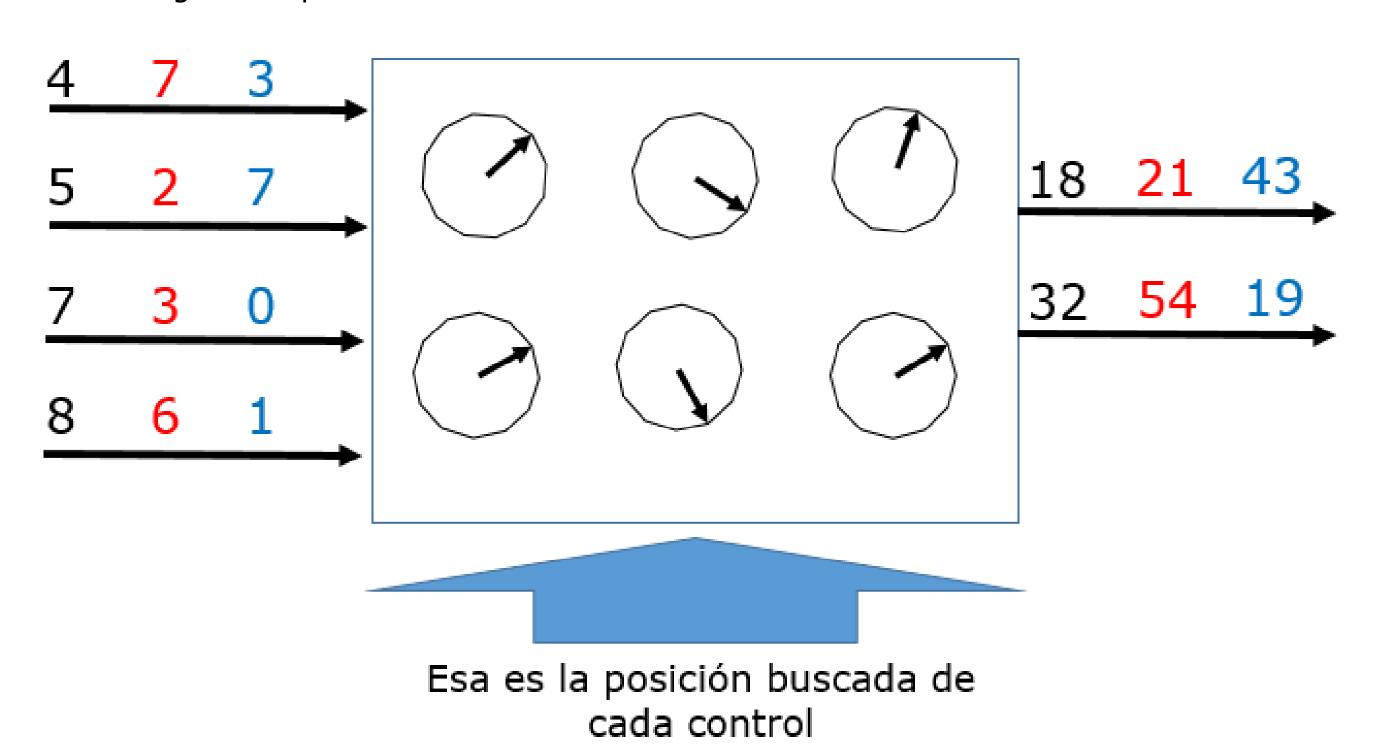
	Entrada 1	Entrada 2	Entrada 3	Entrada 4	Salida deseada 1	Salida deseada 2
Ejemplo 1	4	5	7	8	18	32
Ejemplo 2	7	2	3	6	21	54
Ejemplo 3	3	7	0	1	43	19

Significa que si entran los números 4, 5, 7, 8, (ejemplo 1), debe salir 18 y 32. Luego hay que ajustar esos controles analógicos (moviéndolos en favor o en contra de las manecillas del reloj) hasta que se obtenga esa salida.



Una vez hecho eso, se prueba con las entradas 7, 2, 3, 6 y debe salir 21 y 54. En caso que no funcione con el segundo juego de entradas, se procede a girar de nuevo esos controles y volver a empezar (si, desde el inicio). Así hasta que ajuste con todos los ejemplos. En el caso de la tabla, con los tres conjuntos de entradas que deben dar con las salidas deseadas.

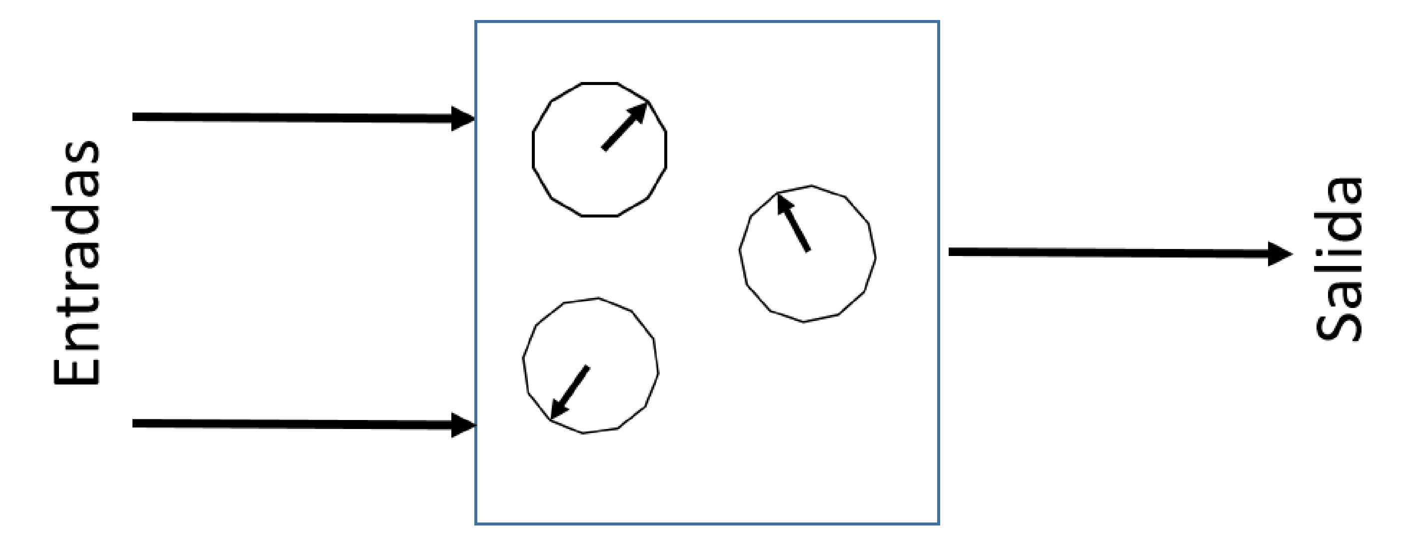
En otras palabras, los 6 controles analógicos deben tener un giro tal, que hace cumplir toda la tabla (los tres ejemplos). El objetivo es dar con esos giros en particular.



¿Y cómo dar con esos giros? Al iniciar, esos controles están girados al azar y poco a poco se van ajustando. Hay un procedimiento matemático que colabora mucho en este caso para así no ajustar a ciegas. Hay que aclarar que la caja tiene 6 controles analógicos, pueden haber muchos más en otras implementaciones.

El “Hola Mundo” de las redes neuronales: El perceptrón simple

Para dar inicio con las redes neuronales se parte de lo más simple: una neurona. Se le conoce como perceptrón simple. Se presenta así:



Dos entradas, una salida y tres controles analógicos. ¿Para qué sirve? Es una demostración que un algoritmo puede aprender la tabla del AND y del OR. Esta es la tabla del AND

Valor A	Valor B	Resultado (A AND B)
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Falso
Falso	Verdadero	Falso
Falso	Falso	Falso

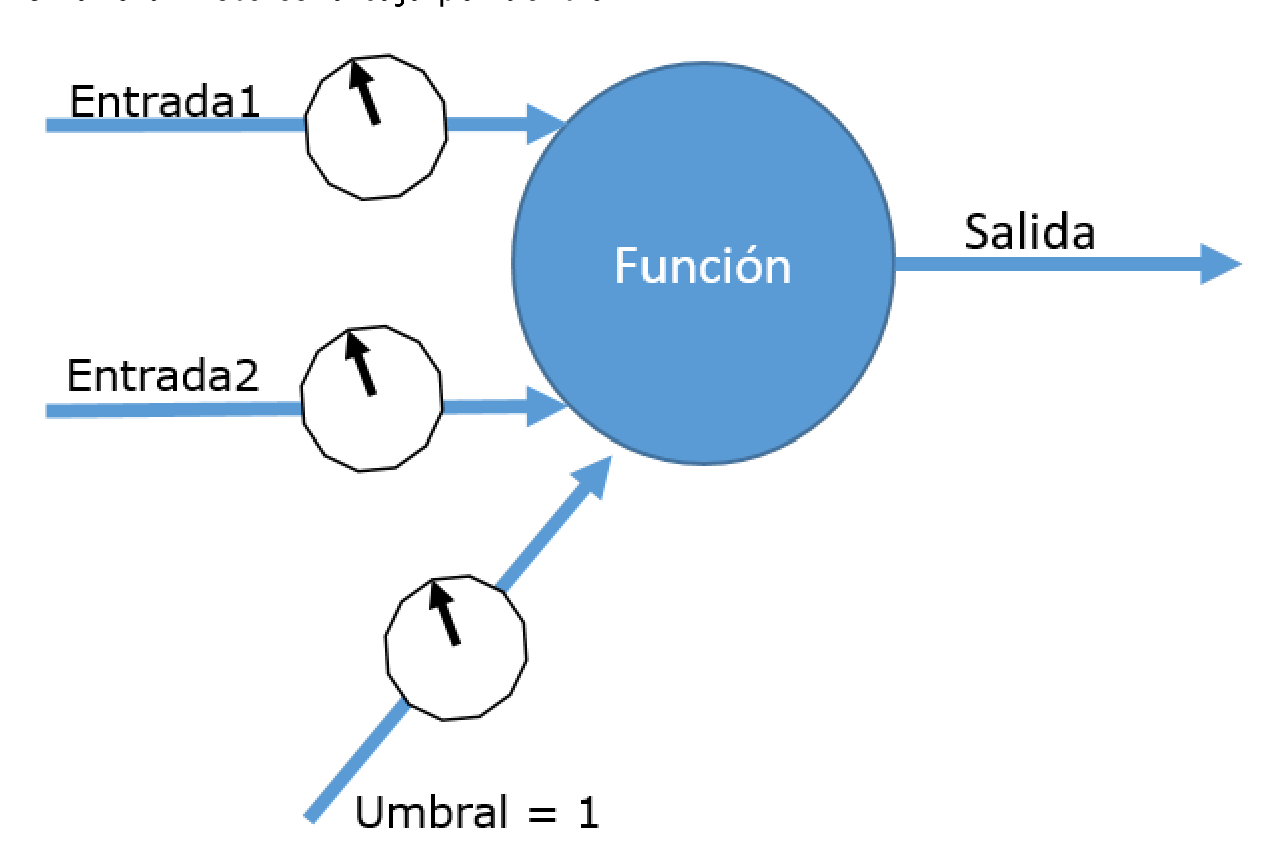
Vamos a hacer que un perceptrón aprenda esa tabla, es decir, que si se ingresa en las entradas Verdadero y Falso, el algoritmo aprenda que debe mostrar en la salida el valor de Falso y así con toda la tabla.

El primer paso es volver cuantitativa esa tabla

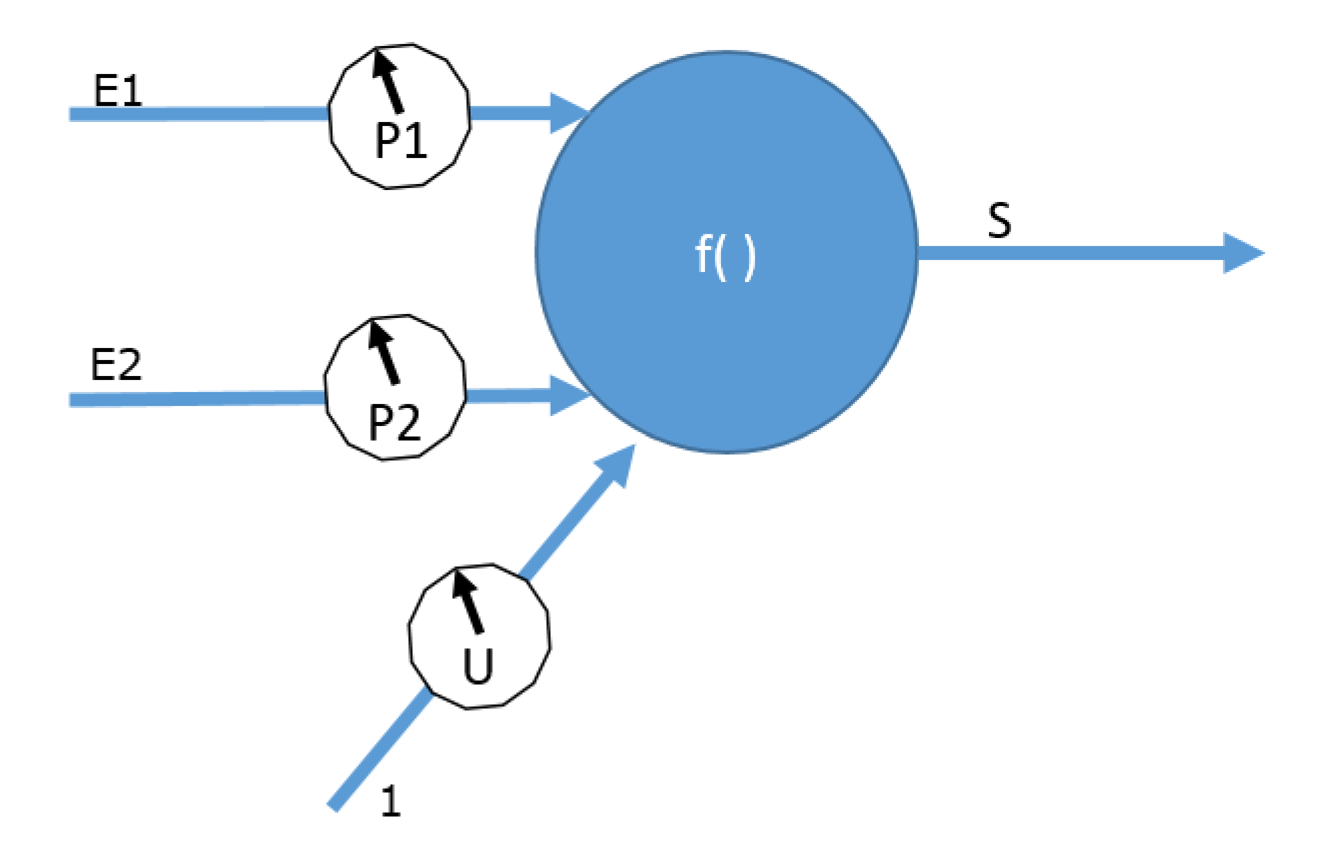
Valor A	Valor B	Resultado (A AND B)
1	1	1
1	0	0
0	1	0
0	0	0

Los datos de entrada y salida deben ser cuantitativos porque en el interior de esa caja hay fórmulas y procedimientos matemáticos. Luego para este ejemplo, 1 representa verdadero y 0 representa falso.

¿Y ahora? Este es la caja por dentro



Un control analógico por cada entrada y se le adiciona una entrada interna que se llama umbral y tiene el valor de 1 con su propio control analógico. Esos controles analógicos se llaman pesos. Ahora se le ponen nombres a cada parte.



E1 y E2 son las entradas

P1, P2 son los pesos de las entradas

U es el peso del umbral

S es la salida

f( ) es la función que le da el valor a S

Luego la salida se calcula así:

S = f (E1 * P1 + E2 * P2 + 1 * U)

Para entenderlo mejor, vamos a darle unos valores:

E1 = 1 (verdadero)
E2 = 1 (verdadero)

P1 = 0.9812 (un valor real al azar)
P2 = 3.7193 (un valor real al azar)
U = 2.1415 (un valor real al azar)

Entonces la salida sería:

S = f (E1 * P1 + E2 * P2 + 1 * P3)

S = f (1 * 0.9812 + 1 * 3.7193 + 1 * 2.1415)

S = f (6.842)

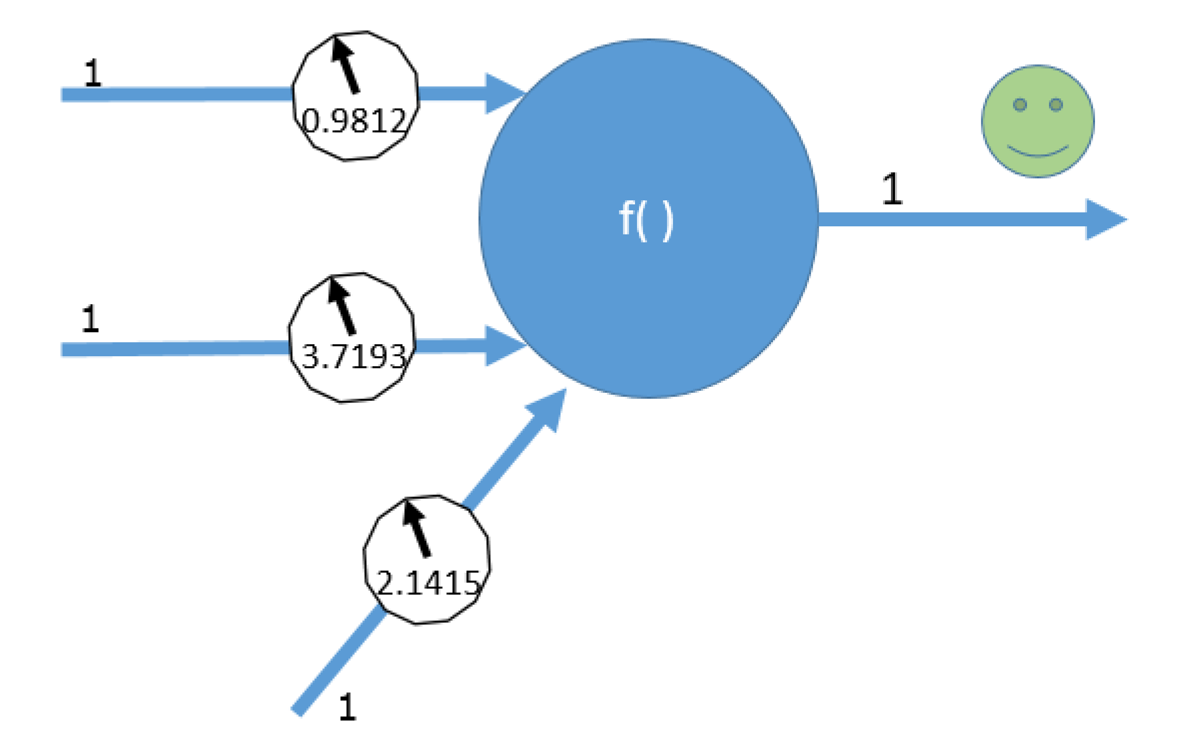
¿Y que es f( )? una función que podría implementarse así:

```
Función f(valor)
Inicio
     Si valor > 0 entonces
             retorne 1
     de lo contrario
             retorne 0
     fin si
Fin
```

Continuando con el ejemplo entonces

S = f (6.842)
S = 1

Y ese es el valor esperado. Los pesos funcionan para esas entradas.



¿Funcionarán esos pesos para las otras entradas? ¡Probemos!

E1 = 1 (verdadero)
E2 = 0 (falso)

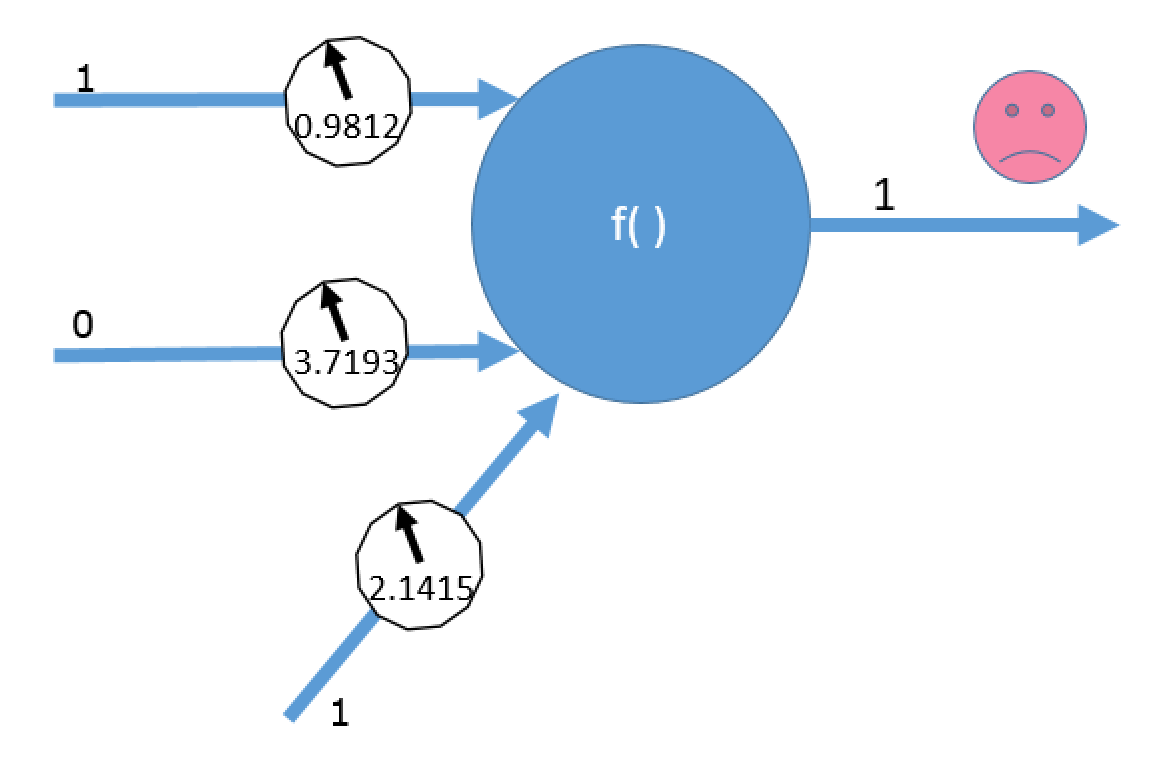
S = f (E1 * P1 + E2 * P2 + 1 * P3)

S = f (1 * 0.9812 + 0 * 3.7193 + 1 * 2.1415)

S = f (3.1227)

S = 1

No, no funcionó, debería haber dado cero



¿Y entonces? Habrá que utilizar otros valores para los pesos. Una forma es darle otros valores al azar. Ejecutar de nuevo el proceso, probar con todas las entradas hasta que finalmente de las salidas esperadas.

Este sería la implementación en C#

```
using System;
namespace Perceptron {
 class Program {
  static void Main(string[] args){
   int[,] datos = { { 1, 1, 1 }, { 1, 0, 0 }, { 0, 1, 0 }, { 0, 0, 0 } }; //Tabla de verdad AND
   Random azar = new Random();
   double[] pesos = { azar.NextDouble(), azar.NextDouble(), azar.NextDouble() }; //Inicia los pesos al azar
   bool aprendiendo = true;
   int salidaEntera;

    while (aprendiendo){ //Hasta que aprenda la tabla AND
     aprendiendo = false;
     for (int cont = 0; cont <= 3 ; cont++){
      double salidaReal = datos[cont, 0] * pesos[0] + datos[cont,1] * pesos[1] + pesos[2]; //Calcula la salida real
      if (salidaReal>0) salidaEntera = 1; else salidaEntera = 0; //Transforma a valores 0 o 1
      if (salidaEntera != datos[cont, 2]) { //Si la salida no coincide con lo esperado, cambia los pesos al azar
       pesos[0] = azar.NextDouble() - azar.NextDouble();
       pesos[1] = azar.NextDouble() - azar.NextDouble();
       pesos[2] = azar.NextDouble() - azar.NextDouble();
       aprendiendo = true; //Y sigue buscando
      }
     }
   }

   for (int cont = 0; cont <= 3; cont++){ //Muestra el perceptron con la tabla AND aprendida
    double salidaReal = datos[cont, 0] * pesos[0] + datos[cont, 1] * pesos[1] + pesos[2];
    if (salidaReal > 0) salidaEntera = 1; else salidaEntera = 0;
    Console.WriteLine("Entradas: " + datos[cont,0].ToString() + " y " + datos[cont,1].ToString() + " = " +
    datos[cont,2].ToString() + " perceptron: " + salidaEntera.ToString());
   }
   Console.ReadLine();
  }
 }
}
```

La línea

```
if (salidaReal>0) salidaEntera = 1; else salidaEntera = 0;
```

Es la función f ( ).

Si los pesos no funcionan entonces se obtienen otros al azar. Cabe anotar que los pesos son reales y pueden ser valores positivos o negativos.

Ejecutando el programa se obtiene:

```
file:///C:/Users/engin/onedrive/documentos/visual studio 2015/Projects/Perceptron/Perceptron/bin/Debug/Perceptron.EXE

Entradas: 1 y 1 = 1 perceptron: 1
Entradas: 1 y 0 = 0 perceptron: 0
Entradas: 0 y 1 = 0 perceptron: 0
Entradas: 0 y 0 = 0 perceptron: 0
```

Se modifica el programa para que muestre los pesos y la cantidad de iteraciones que hizo

```
using System;
namespace Perceptron {
 class Program {
  static void Main(string[] args){
   int[,] datos = { { 1, 1, 1 }, { 1, 0, 0 }, { 0, 1, 0 }, { 0, 0, 0 } }; //Tabla de verdad AND
   Random azar = new Random();
   double[] pesos = { azar.NextDouble(), azar.NextDouble(), azar.NextDouble() }; //Inicia los pesos al azar
   bool aprendiendo = true;
   int salidaEntera, iteracion = 0;

    while (aprendiendo){ //Hasta que aprenda la tabla AND
     iteracion++;
     aprendiendo = false;
     for (int cont = 0; cont <= 3 ; cont++){
      double salidaReal = datos[cont, 0] * pesos[0] + datos[cont,1] * pesos[1] + pesos[2]; //Calcula la salida real
      if (salidaReal>0) salidaEntera = 1; else salidaEntera = 0; //Transforma a valores 0 o 1
      if (salidaEntera != datos[cont, 2]) { //Si la salida no coincide con lo esperado, cambia los pesos al azar
       pesos[0] = azar.NextDouble() - azar.NextDouble();
       pesos[1] = azar.NextDouble() - azar.NextDouble();
       pesos[2] = azar.NextDouble() - azar.NextDouble();
       aprendiendo = true; //Y sigue buscando
      }
     }
   }

   Console.WriteLine("Iteraciones: " + iteracion.ToString());
   Console.WriteLine("Peso 1: " + pesos[0].ToString());
   Console.WriteLine("Peso 2: " + pesos[1].ToString());
   Console.WriteLine("Peso 3: " + pesos[2].ToString());

   for (int cont = 0; cont <= 3; cont++){ //Muestra el perceptron con la tabla AND aprendida
    double salidaReal = datos[cont, 0] * pesos[0] + datos[cont, 1] * pesos[1] + pesos[2];
    if (salidaReal > 0) salidaEntera = 1; else salidaEntera = 0;
    Console.WriteLine("Entradas: " + datos[cont,0].ToString() + " y " + datos[cont,1].ToString() + " = " +
     datos[cont,2].ToString() + " perceptron: " + salidaEntera.ToString());
   }
   Console.ReadLine();
  }
 }
}
```

Este es el resultado. Nota: Peso 3 es el peso del umbral

```
file:///C:/Users/engin/onedrive/documentos/visual studio 2015/Projects/Perceptron/Perceptron/bin/Debug/Perceptron.EXE
Iteraciones: 6
Peso 1: 0,0490536201042373
Peso 2: 0,385779543493772
Peso 3: -0,416483992438989
Entradas: 1 y 1 = 1 perceptron: 1
Entradas: 1 y 0 = 0 perceptron: 0
Entradas: 0 y 1 = 0 perceptron: 0
Entradas: 0 y 0 = 0 perceptron: 0
```

Y volviendo a ejecutar

```
file:///C:/Users/engin/onedrive/documentos/visual studio 2015/Projects/Perceptron/Perceptron/bin/Debug/Perceptron.EXE
Iteraciones: 29
Peso 1: 0,594828198009556
Peso 2: 0,277698465752275
Peso 3: -0,762235941254644
Entradas: 1 y 1 = 1 perceptron: 1
Entradas: 1 y 0 = 0 perceptron: 0
Entradas: 0 y 1 = 0 perceptron: 0
Entradas: 0 y 0 = 0 perceptron: 0
```

Observamos que los pesos no es una respuesta única, pueden ser distintos y son números reales (en C# se implementaron de tipo double), esa es la razón por la que se llama en este libro controles análogos. También observamos que en una ejecución requirió sólo 6 iteraciones y en la siguiente ejecución requirió 29 iteraciones. El cambio de pesos sucede en estas líneas:

```
       pesos[0] = azar.NextDouble() - azar.NextDouble();
       pesos[1] = azar.NextDouble() - azar.NextDouble();
       pesos[2] = azar.NextDouble() - azar.NextDouble();
```

En caso de que no funcionasen los pesos, el programa simplemente los cambiaba al azar en un valor que oscila entre -1 y 1. Eso puede ser muy ineficiente y riesgoso porque limita los valores a estar entre -1 y 1 ¿Y si los pesos requieren valores mucho más altos o más bajos?

Afortunadamente, hay un método matemático que minimiza el uso del azar y puede dar con valores de los pesos en cualquier rango. ¿Cómo funciona? Al principio los pesos tienen un valor al azar, pero de allí en adelante el cálculo de esos pesos se basa en comparar la salida esperada con la salida obtenida, si difieren, ese error sirve para ir cuadrando poco a poco los pesos.

Fórmula de Frank Rosenblatt

En vez de cambiar los pesos en forma aleatoria, se hace uso de una serie de fórmulas matemáticas.

```
Error = Salida Esperada – Salida Real
Si Error es diferente de cero entonces
      Nuevo Peso (para entrada 1) = Peso anterior (para entrada 1) + tasa aprende * Error * Entrada 1
      Nuevo Peso (para entrada 2) = Peso anterior (para entrada 2) + tasa aprende * Error * Entrada 2
      Nuevo Peso (para umbral) = Peso anterior (de la entrada del umbral) + tasa aprende * Error * 1
Fin Si
```

Tasa aprende es un valor constante de tipo real y de valor entre 0 y 1 (sin tomar el 0, ni el 1)

Este es el código en C#

```
using System;
namespace Perceptron2 {
  public class Program {
      public static void Main(String[] args){
        int[,] tabla = { { 1, 1, 1 }, { 1, 0, 0 }, { 0, 1, 0 }, { 0, 0, 0 } }; //Tabla de verdad AND: { x1, x2, salida }
        Random azar = new Random();
        double[] pesos = { azar.NextDouble(), azar.NextDouble(), azar.NextDouble() }; //Inicia los pesos al azar
        bool aprendiendo = true;
        int salidaEntera, iteracion = 0;
        double tasaAprende = 0.3;
        while (aprendiendo) { //Hasta que aprenda la tabla AND
          iteracion++;
          aprendiendo = false;
          for (int cont = 0; cont <= 3; cont++) {
            double salidaReal = tabla[cont, 0] * pesos[0] + tabla[cont, 1] * pesos[1] + pesos[2]; //Calcula la salida real
            if (salidaReal > 0) salidaEntera = 1; else salidaEntera = 0; //Transforma a valores 0 o 1
            int error = tabla[cont, 2] - salidaEntera;
            if (error != 0){ //Si la salida no coincide con lo esperado, cambia los pesos con la fórmula de Frank Rosenblatt
                pesos[0] += tasaAprende * error * tabla[cont, 0];
                pesos[1] += tasaAprende * error * tabla[cont, 1];
                pesos[2] += tasaAprende * error * 1;
                aprendiendo = true; //Y sigue buscando
            }
          }
        }

        Console.WriteLine("Iteraciones: " + iteracion.ToString());
        Console.WriteLine("Peso 1: " + pesos[0].ToString());
        Console.WriteLine("Peso 2: " + pesos[1].ToString());
        Console.WriteLine("Peso 3: " + pesos[2].ToString());

        for (int cont = 0; cont <= 3; cont++){ //Muestra el perceptron con la tabla AND aprendida
           double salidaReal = tabla[cont, 0] * pesos[0] + tabla[cont, 1] * pesos[1] + pesos[2];
           if (salidaReal > 0) salidaEntera = 1; else salidaEntera = 0;
            Console.WriteLine(tabla[cont, 0] + " y " + tabla[cont, 1] + " = " + tabla[cont, 2] + " perceptron: " +
salidaEntera);
        }
        Console.ReadKey();
      }
  }
}
```

Así ejecuta el programa

```
file:///C:/Users/engin/onedrive/documentos/visual studio 2015/Projects/Perceptron2/Perceptron2/bin/Debug/Perceptron2.EXE
Iteraciones: 8
Peso 1: 0,543821145614526
Peso 2: 0,840766710201635
Peso 3: -1,09279940998778
1 y 1 = 1 perceptron: 1
1 y 0 = 0 perceptron: 0
0 y 1 = 0 perceptron: 0
0 y 0 = 0 perceptron: 0
```

Una vez más se ejecuta

```
file:///C:/Users/engin/onedrive/documentos/visual studio 2015/Projects/Perceptron2/Perceptron2/bin/Debug/Perceptron2.EXE
Iteraciones: 2
Peso 1: 0,256831110993787
Peso 2: 0,349154528812112
Peso 3: -0,399396705720293
1 y 1 = 1 perceptron: 1
1 y 0 = 0 perceptron: 0
0 y 1 = 0 perceptron: 0
0 y 0 = 0 perceptron: 0
```

Y ese es el aprendizaje, un ajuste de pesos o constantes a una serie de ecuaciones hasta dar con las salidas requeridas para todas las entradas.

¿Y si se varían los valores que representan el verdadero y falso? ¿Variar la función?

Podríamos discutir que fue conveniente haber puesto "0" a falso y "1" a verdadero, ¿se podrían otros valores? Es cuestión de probar

Valor A	Valor B	Resultado (A AND B)
5	5	5
5	-3	-3
-3	5	-3
-3	-3	-3

El código sólo varía así:

```
using System;
namespace Perceptron2 {
    public class Program {
        public static void Main(String[] args){
            int[,] tabla = { { 5, 5, 5 }, { 5, -3, -3 }, { -3, 5, -3 }, { -3, -3, -3 } }; //Tabla de verdad AND: { x1, x2, salida }
            Random azar = new Random();
            double[] pesos = { azar.NextDouble(), azar.NextDouble(), azar.NextDouble() }; //Inicia los pesos al azar
            bool aprendiendo = true;
            int salidaEntera, iteracion = 0;
            double tasaAprende = 0.3;
            while (aprendiendo) { //Hasta que aprenda la tabla AND
                iteracion++;
                aprendiendo = false;
                for (int cont = 0; cont <= 3; cont++) {
                    double salidaReal = tabla[cont, 0] * pesos[0] + tabla[cont, 1] * pesos[1] + pesos[2]; //Calcula la salida real
                    if (salidaReal > 0) salidaEntera = 5; else salidaEntera = -3; //Transforma a valores 5 o -3
                    int error = tabla[cont, 2] - salidaEntera;
                    if (error != 0){ //Si la salida no coincide con lo esperado, cambia los pesos con la fórmula de Frank Rosenblatt
                        pesos[0] += tasaAprende * error * tabla[cont, 0];
                        pesos[1] += tasaAprende * error * tabla[cont, 1];
                        pesos[2] += tasaAprende * error * 1;
                        aprendiendo = true; //Y sigue buscando
                    }
                }
            }

            Console.WriteLine("Iteraciones: " + iteracion.ToString());
            Console.WriteLine("Peso 1: " + pesos[0].ToString());
            Console.WriteLine("Peso 2: " + pesos[1].ToString());
            Console.WriteLine("Peso 3: " + pesos[2].ToString());

            for (int cont = 0; cont <= 3; cont++){ //Muestra el perceptron con la tabla AND aprendida
                double salidaReal = tabla[cont, 0] * pesos[0] + tabla[cont, 1] * pesos[1] + pesos[2];
                if (salidaReal > 0) salidaEntera = 5; else salidaEntera = -3;
                Console.WriteLine(tabla[cont, 0] + " y " + tabla[cont, 1] + " = " + tabla[cont, 2] + " perceptron: " + salidaEntera);
            }
            Console.ReadKey();
        }
    }
}
```

Al ejecutar

```
file:///C:/Users/engin/onedrive/documentos/visual studio 2015/Projects/Perceptron2/Perceptron2/bin/Debug/Perceptron2.EXE
Iteraciones: 3
Peso 1: 2,45695001038581
Peso 2: 2,77383257102865
Peso 3: -7,02539789696475
5 y 5 = 5 perceptron: 5
5 y -3 = -3 perceptron: -3
-3 y 5 = -3 perceptron: -3
-3 y -3 = -3 perceptron: -3
```

El resultado es el mismo, el perceptrón aprende.

¿Y cambiar la función? En este caso en particular, el resultado sólo es uno de dos posibles valores, luego el cambio sería por el sí condicional que compare con otro valor

```
if (salidaReal > 1) salidaEntera = 5; else salidaEntera = -3; //Transforma a valores 5 o -3
```

Se obtiene un resultado de aprendizaje correcto

```
file:///C:/Users/engin/onedrive/documentos/visual studio 2015/Projects/Perceptron2/Perceptron2/bin/Debug/Perceptron2.EXE
Iteraciones: 2
Peso 1: 3,07136371958599
Peso 2: 3,07284173270354
Peso 3: -6,30233852830824
5 y 5 = 5 perceptron: 5
5 y -3 = -3 perceptron: -3
-3 y 5 = -3 perceptron: -3
-3 y -3 = -3 perceptron: -3
```

Perceptrón simple: Aprendiendo la tabla del OR

El ejemplo anterior el perceptrón simple aprendía la tabla AND, ¿y con la OR?

Valor A	Valor B	Resultado (A OR B)
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Verdadero
Falso	Verdadero	Verdadero
Falso	Falso	Falso

Vamos a hacer que el perceptrón aprenda esa tabla, es decir, que si se ingresa en las entradas Verdadero y Falso, el perceptrón aprenda que debe mostrar en la salida el valor de Falso y así con toda la tabla.

El primer paso es volver cuantitativa esa tabla

Valor A	Valor B	Resultado (A OR B)
1	1	1
1	0	1
0	1	1
0	0	0

Es sólo cambiar esta línea del programa

```
int[,] tabla = { { 1, 1, 1 }, { 1, 0, 0 }, { 0, 1, 0 }, { 0, 0, 0 } }; //Tabla de verdad AND: { x1, x2, salida }
```

por esta

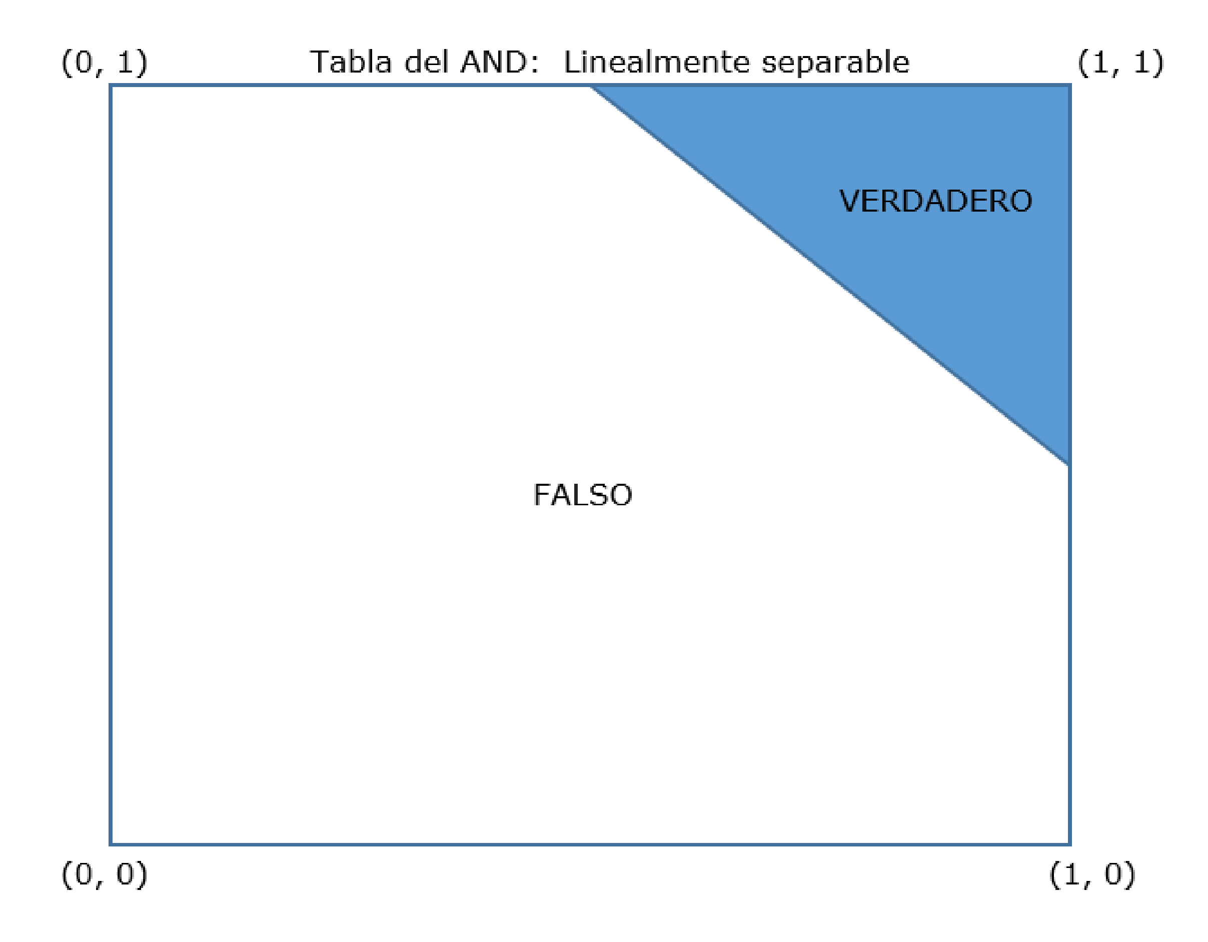
```
int[,] tabla = { { 1, 1, 1 }, { 1, 0, 1 }, { 0, 1, 1 }, { 0, 0, 0 } }; //Tabla de verdad OR: { x1, x2, salida }
```

Y volver a ejecutar la aplicación

```
file:///C:/Users/engin/onedrive/documentos/visual studio 2015/Projects/Perceptron2/Perceptron2/bin/Debug/Perceptron2.EXE
Iteraciones: 3
Peso 1: 0,627607005009245
Peso 2: 0,547981312287963
Peso 3: -0,260314471768362
1 y 1 = 1 perceptron: 1
1 y 0 = 1 perceptron: 1
0 y 1 = 1 perceptron: 1
0 y 0 = 0 perceptron: 0
```

Límites del Perceptrón Simple

El perceptrón simple tiene un límite: que sólo sirve cuando la solución se puede separar con **una** recta. Se explica a continuación

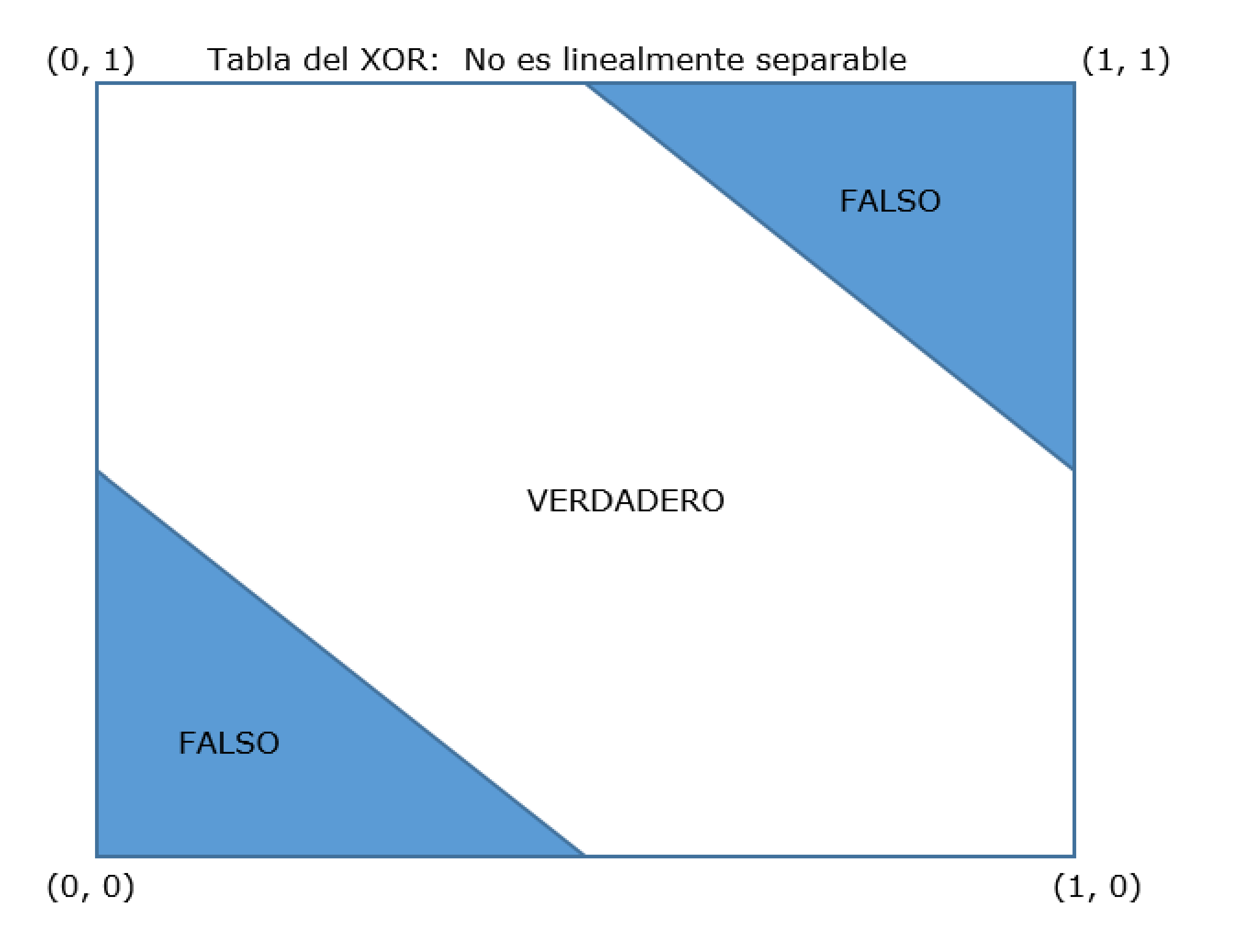


En cambio, si se quiere abordar un problema que requiera dos separaciones, no lo podría hacer el perceptrón simple. El ejemplo clásico es la tabla XOR

Valor A	Valor B	Resultado (A XOR B)
Verdadero	Verdadero	Falso
Verdadero	Falso	Verdadero
Falso	Verdadero	Verdadero
Falso	Falso	Falso

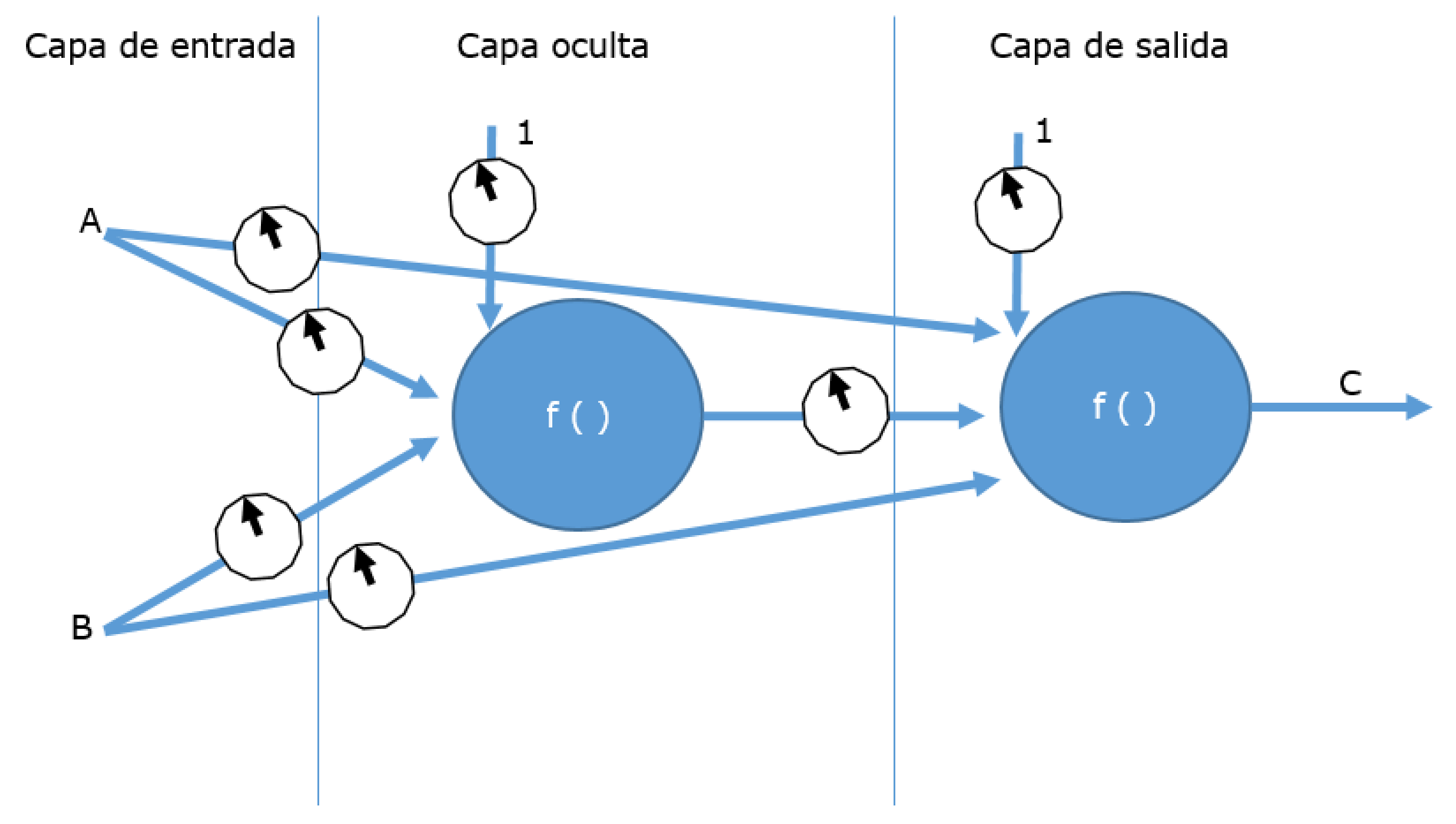
Cuantitativa esa tabla

Valor A	Valor B	Resultado (A XOR B)
1	1	0
1	0	1
0	1	1
0	0	0

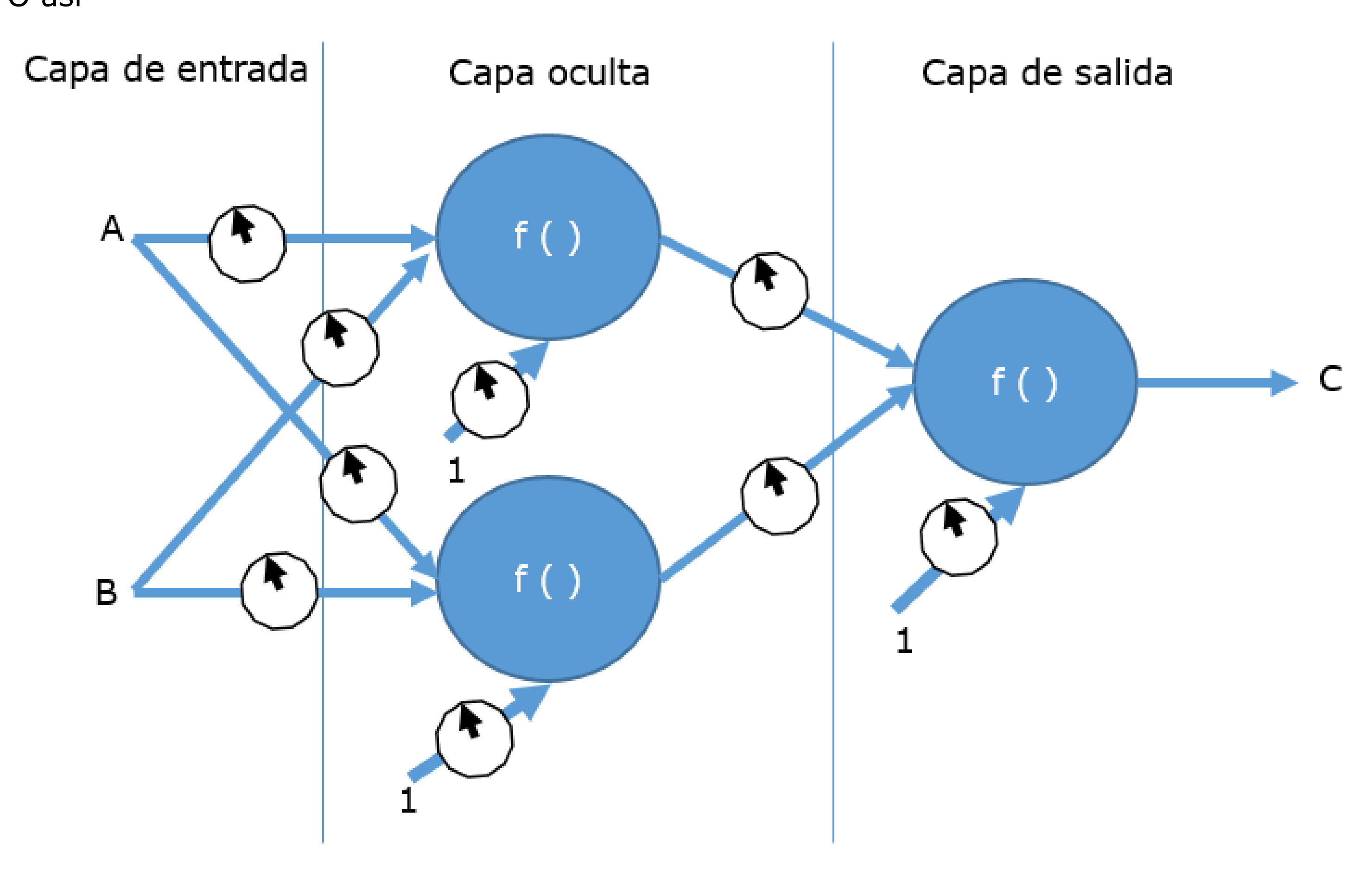


¿Y qué se puede hacer allí?

Se necesita entonces varias neuronas más y puestas en capas.



O así



Varios controles analógicos o pesos, el reto es cómo dar con el peso correcto para cada control. Hay entonces un estudio matemático para ello.

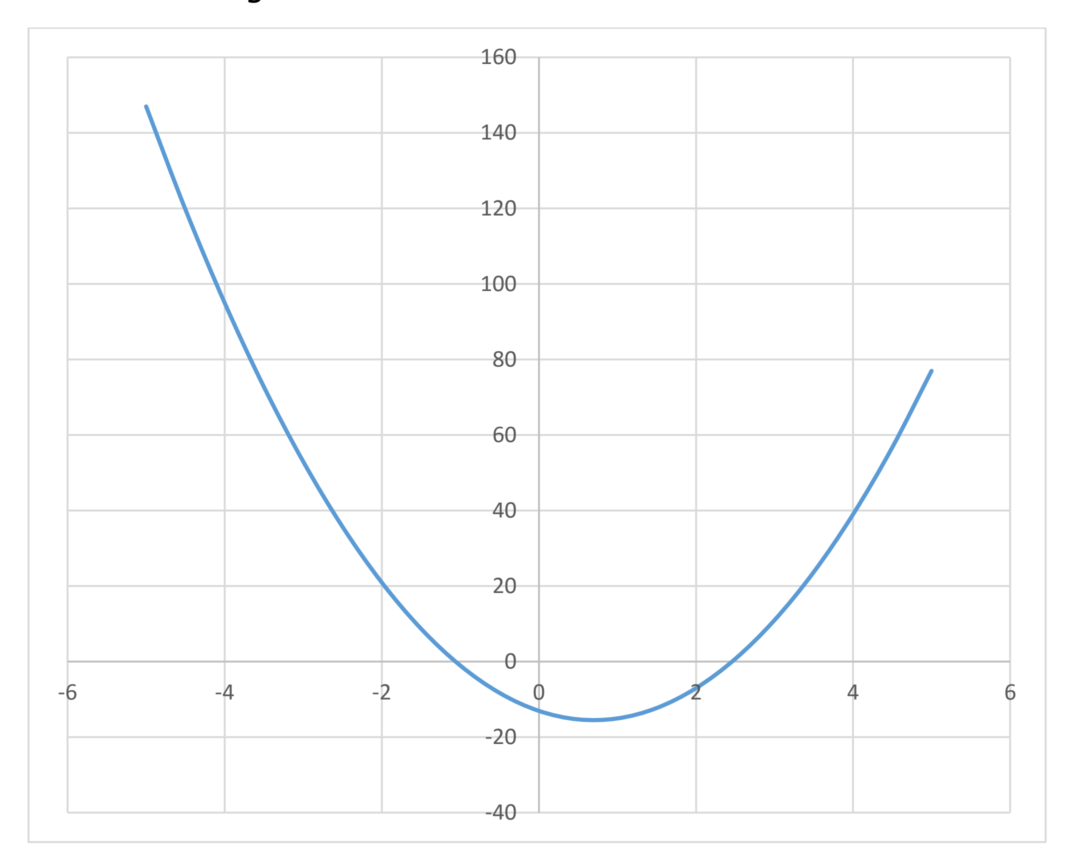
Encontrando el mínimo en una ecuación

Empecemos con la base matemática que nos ayudará a deducir los pesos en una red neuronal.

Para dar con el mínimo de una ecuación se hace uso de las derivadas. Un ejemplo: tenemos la ecuación

$$y = 5 * x^2 - 7 * x - 13$$

Este sería su gráfico



Si queremos dar con el valor de x para que y sea el mínimo valor, el primer paso es derivar

$$y' = 2 * 5 * x - 7$$

Luego esa derivada se iguala a cero

$$0 = 2 * 5 * x - 7$$

Se resuelve el valor de x

$$0 = 10 * x - 7$$

$$x = 7/10$$

$$x = 0.7$$

Y tenemos el valor de x con el que se obtiene el mínimo valor de y

$$y = 5 * x^2 - 7 * x - 13$$

$$y = 5 * 0.7^2 - 7 * 0.7 - 13$$

$$y = -15.45$$

En este caso fue fácil dar con la derivada, porque fue un polinomio de grado 2, el problema sucede cuando la ecuación es compleja, derivarla se torna un desafío y despejar x sea muy complicado.

Otra forma de dar con el mínimo es iniciar con algún punto x al azar, por ejemplo, x = 1.0

Valor de X	$y = 5 * x^2 - 7 * x - 13$
1.0	-15

Bien, ahora nos desplazamos, tanto a la izquierda como a la derecha de 0.5 en 0.5, es decir, x=0.5 y x=1.5

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.5	-15.25
1.0	**-15**
1.5	-12,25

Ya tenemos un nuevo valor de X más prometedor que es 0.5, luego se repite el procedimiento, izquierda y derecha, es decir, x=0.0 y x=1.0

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.0	-13
0.5	**-15.25**
1.0	-15

El valor de 0.5 se mantiene como el mejor, luego se hace izquierda y derecha a un paso menor de 0.25

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.25	-14,4375
0.50	**-15.25**
0.75	-15.4375

El valor de x=0.75 es el que muestra mejor comportamiento, luego se hace izquierda y derecha a un paso de 0.25

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.50	-15.25
0.75	**-15.4375**
1.00	-15

Sigue x=0.75 como mejor valor, luego se prueba a izquierda y derecha pero en una variación menor de 0.125

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.625	-15.421875
0.75	**-15.4375**
0.875	-15.296875

Sigue x=0.75 como mejor valor, luego se prueba a izquierda y derecha pero en una variación menor de 0.0625

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.6875	-15.4492188
0.75	**-15.4375**
0.8125	-15.3867188

Ahora es x=0.6875 como mejor valor, luego se prueba a izquierda y derecha en una variación de 0.0625

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.625	-15.421875
0.6875	**-15.4492188**
0,75	-15.4375

Sigue x=0.6875 como mejor valor, luego se prueba a izquierda y derecha pero en una variación menor de 0.03125

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.65625	-15.4404297
0.6875	**-15.4492188**
0,71875	-15.4482422

Sigue x=0.6875 como mejor valor, luego se prueba a izquierda y derecha pero en una variación menor de 0.015625

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.671875	-15.4460449
0.6875	**-15.4492188**
0,703125	-15.4499512

Ahora es x=0,703125 como mejor valor. Como podemos observar, ese método se aproxima a x=0.7 que es el resultado que se dedujo con las derivadas.

Otra forma de hacerlo es con el siguiente algoritmo implementado en C#

```
using System;

namespace Minimo {
    class Program {
        static void Main(string[] args)
        {
            double x = 1; //valor inicial
            double Yini = Ecuacion(x);
            double variacion = 1;

            while (Math.Abs(variacion) > 0.00001)
            {
                double Ysigue = Ecuacion(x+variacion);
                if (Ysigue > Yini){ //Si no disminuye, cambia de dirección a un paso menor
                    variacion *= -1;
                    variacion /= 10;
                }
                else {
                    Yini = Ysigue; //Disminuye
                    x += variacion;
                    Console.WriteLine("x: " + x.ToString() + " Yini:" + Yini.ToString());
                }
            }
            Console.WriteLine("Respuesta: " + x.ToString());
            Console.ReadKey();
        }

        static double Ecuacion(double x) {
            return 5 * x * x - 7 * x - 13;
        }
    }
}
```

Y así ejecuta

```
file:///C:/Users/engin/OneDrive/Documentos/Visual Studio 2015/Projects/
x: 0,9 Yini:-15,25
x: 0,8 Yini:-15,4
x: 0,7 Yini:-15,45
Respuesta: 0,7
```

Y cambiando el valor inicial de x a un valor x=1.13 por ejemplo, esto pasaría:

```
file:///C:/Users/engin/OneDrive/Document
x: 1,03 Yini:-14,9055
x: 0,93 Yini:-15,1855
x: 0,83 Yini:-15,3655
x: 0,73 Yini:-15,4455
x: 0,729 Yini:-15,445795
x: 0,728 Yini:-15,44608
x: 0,727 Yini:-15,446355
x: 0,726 Yini:-15,44662
x: 0,725 Yini:-15,446875
x: 0,724 Yini:-15,44712
x: 0,723 Yini:-15,447355
x: 0,722 Yini:-15,44758
x: 0,721 Yini:-15,447795
x: 0,72 Yini:-15,448
x: 0,719 Yini:-15,448195
x: 0,718 Yini:-15,44838
x: 0,717 Yini:-15,448555
x: 0,716 Yini:-15,44872
x: 0,715 Yini:-15,448875
x: 0,714 Yini:-15,44902
x: 0,713 Yini:-15,449155
x: 0,712 Yini:-15,44928
x: 0,711 Yini:-15,449395
x: 0,71 Yini:-15,4495
x: 0,709 Yini:-15,449595
x: 0,708 Yini:-15,44968
x: 0,707 Yini:-15,449755
x: 0,706 Yini:-15,44982
x: 0,705 Yini:-15,449875
x: 0,704 Yini:-15,44992
```

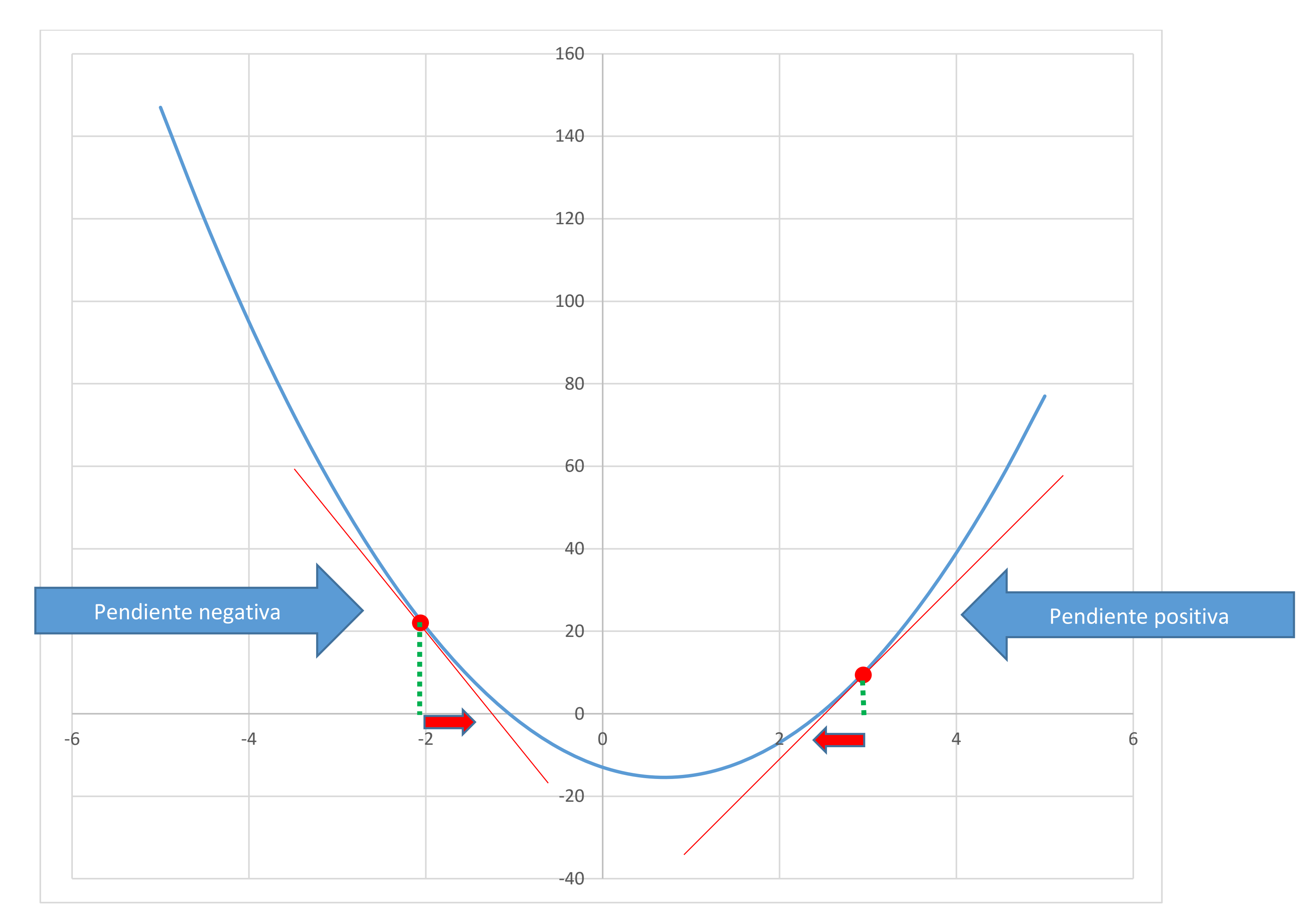
Descenso del gradiente

Anteriormente vimos, con las aproximaciones, como buscar el mínimo valor de Y modificando el valor de X, ya sea yendo por la izquierda (disminuyendo) o por la derecha (aumentando). Matemáticamente para saber en qué dirección ir, es con esta expresión:

$$\Delta x = -y'$$

¿Qué significa? Que x debe modificarse en contra de la derivada de la ecuación.

¿Por qué? La derivada nos muestra la tangente que pasa por el punto que se seleccionó al azar al inicio. Esa tangente es una línea recta y como toda línea recta tiene una pendiente. Si la pendiente es positiva entonces X se debe ir hacia la izquierda (el valor de X debe disminuir), en cambio, si la pendiente es negativa entonces X debe ir hacia la derecha (el valor de X debe aumentar). Con esa indicación ya sabemos por dónde ir para dar con el valor de X que obtiene el mínimo Y.



Para dar con el nuevo valor de X esta sería la expresión:

$$x_{nuevo} = x_{anterior} + \Delta x$$

Reemplazando

$$x_{nuevo} = x_{anterior} + -y'$$

Simplificando

$$x_{nuevo} = x_{anterior} - y'$$

EJEMPLO

Con la ecuación anterior

$$y = 5 * x^2 - 7 * x - 13$$

$$y' = 10 * x - 7$$

X inicia en 1, luego

$$x_{nuevo} = x_{anterior} - y'$$

$$x_{nuevo} = x_{anterior} - (10 * x - 7)$$

¿Y esa x? Por supuesto que es la anterior porque estamos variando

$$x_{nuevo} = x_{anterior} - (10 * x_{anterior} - 7)$$

$$x_{nuevo} = 1 - (10 * 1 - 7)$$

$$x_{nuevo} = -2$$

Ahora hay un nuevo valor para X que es -2. En la siguiente tabla se muestra como progresa X

X anterior	Valor Y	X nuevo
1	-15	-2
-2	21	25
25	2937	-218
-218	239133	1969
1969	19371009	-17714
-17714	1569052965	159433
159433	1,2709E+11	-1434890

El valor de X se dispara, se vuelve extremo hacía la izquierda o derecha. Podríamos concluir que el método falla estrepitosamente. No tan rápido, se puede arreglar y es agregando una constante a la ecuación

$$x_{nuevo} = x_{anterior} - \propto * y'$$

Se agrega entonces un α que es una constante muy pequeña, por ejemplo α=0,05 y esto es lo que sucede

$$x_{nuevo} = x_{anterior} - 0{,}05 * (10 * x_{anterior} - 7)$$

X anterior	Valor Y	X nuevo
1	-15	0,85
0,85	-15,3375	0,775
0,775	-15,421875	0,7375
0,7375	-15,4429688	0,71875
0,71875	-15,4482422	0,709375
0,709375	-15,4495605	0,7046875
0,7046875	-15,4498901	0,70234375

Tiene más sentido y se acerca a X=0.7 que es la respuesta correcta.

Este método se le conoce como el descenso del gradiente que se expresa así

$$x_{nuevo} = x_{anterior} - \propto * f'(x_{anterior})$$

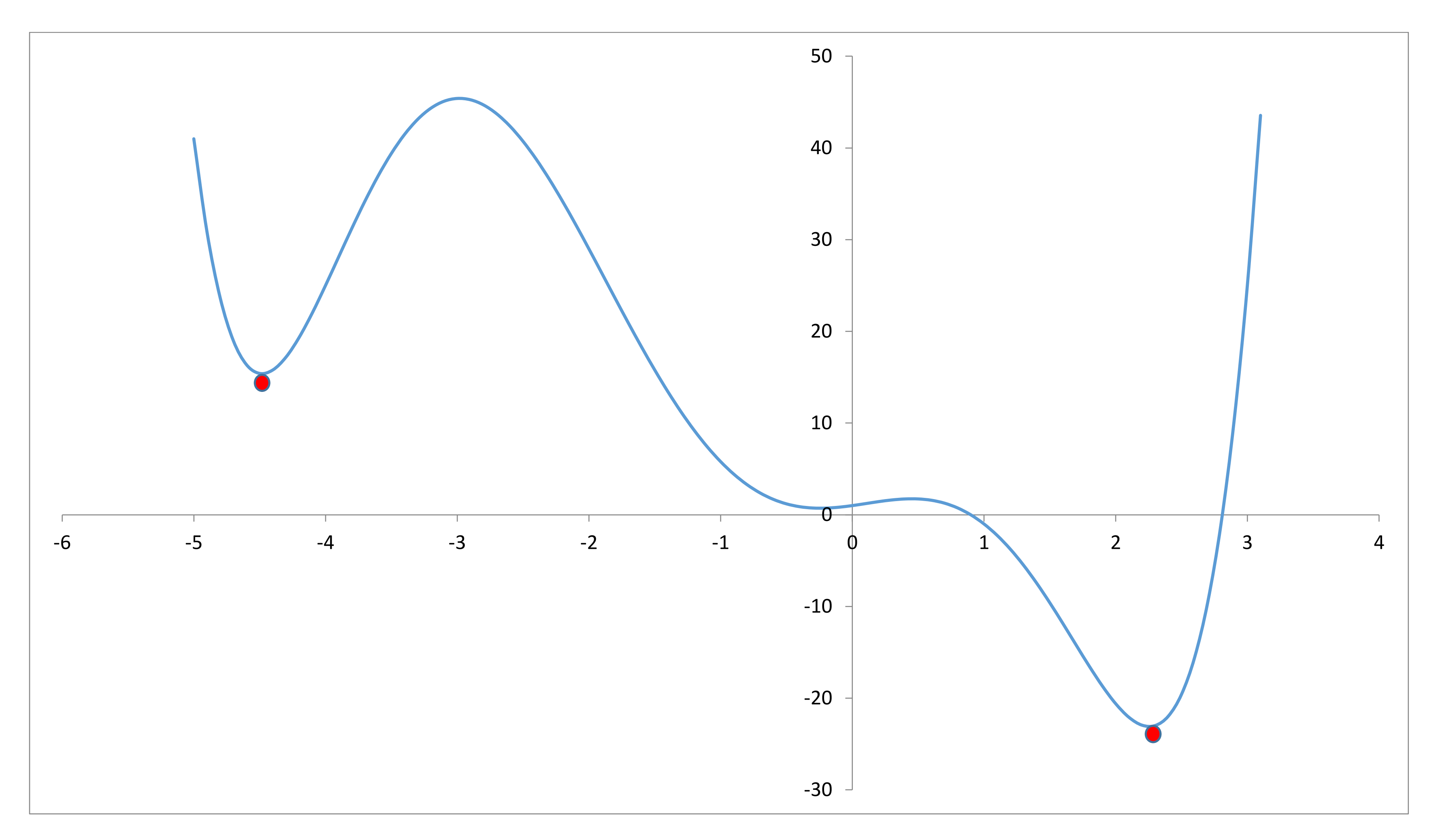
En formato clásico matemático

$$x_{n+1} = x_n - \propto * f'(x_n)$$

A tener en cuenta en la búsqueda de mínimos

La siguiente curva es generada por el siguiente polinomio

$$y = 0.1 * x^6 + 0.6 * x^5 - 0.7 * x^4 - 6 * x^3 + 2 * x^2 + 2 * x + 1$$



Se aprecian dos puntos donde claramente la curva desciende y vuelve a ascender (se han marcado con puntos en rojo), por supuesto, el segundo a la derecha es el mínimo real, pero, ¿Qué pasaría si se hubiese hecho una búsqueda iniciando en x=-4? La respuesta es que el algoritmo se hubiese decantado por el mínimo de la izquierda. Veamos:

$$x_{nuevo} = x_{anterior} - \propto * \, y'$$

$$x_{nuevo} = x_{anterior} - 0{,}01 * (0.6 * x^5 + 3 * x^4 - 2.8 * x^3 - 18 * x^2 + 4 * x + 2)$$

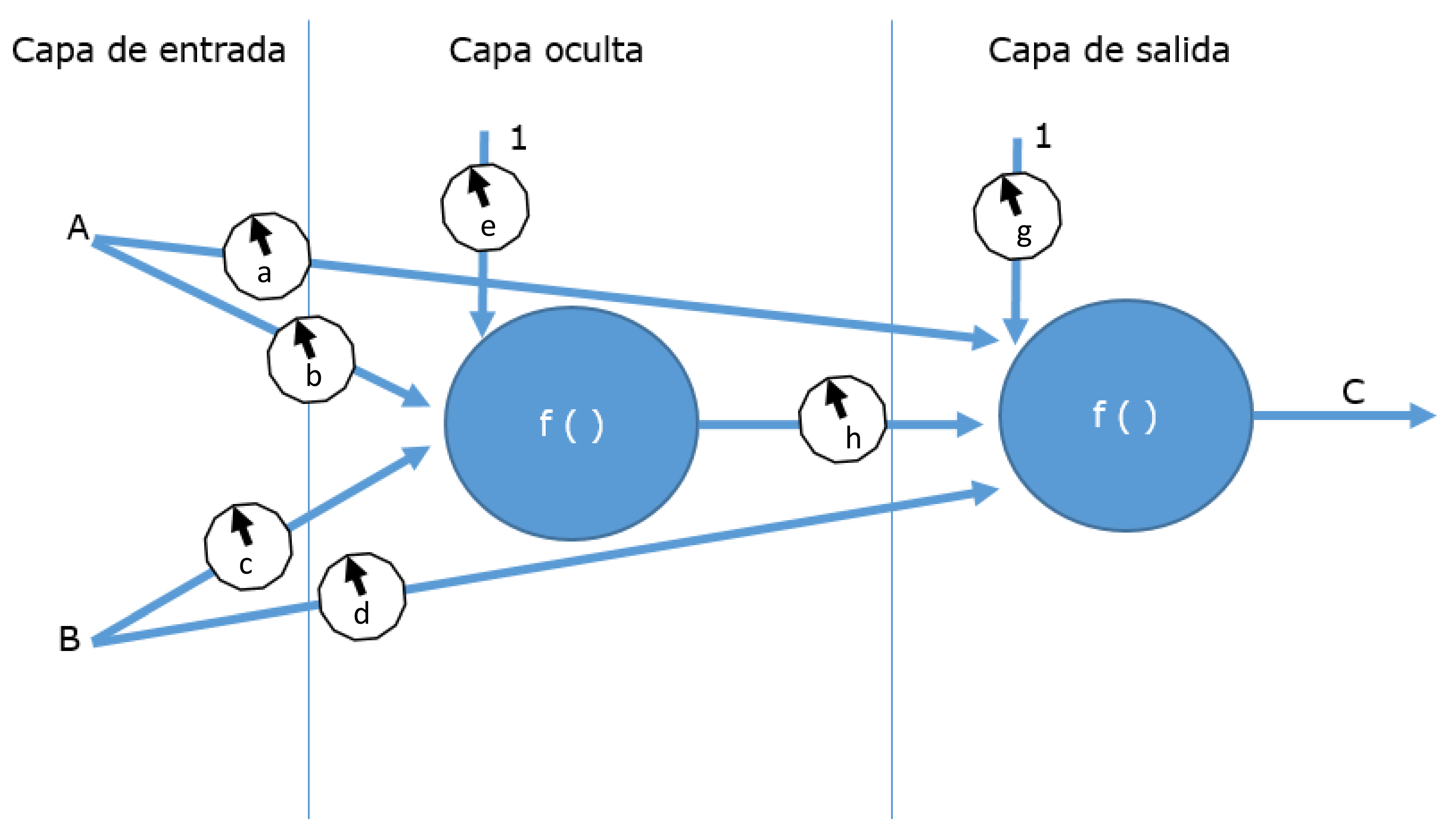
Xanterior	Y	Xnuevo
-4	25	-4,308
-4,308	17,0485	-4,4838
-4,4838	15,3935	-4,4815
-4,4815	15,3933	-4,4822
-4,4822	15,3933	-4,482
-4,482	15,3933	-4,482

Este problema se le conoce como caer en mínimo local y también lo sufren los algoritmos genéticos. Así que se deben probar otros valores de X para iniciar, si fuese X=2 observamos que si acierta con el mínimo real

Xanterior	Y	Xnuevo
2	-20,6	2,172
2,172	-22,777	2,24349
2,24349	-23,08	2,25489
2,25489	-23,087	2,25559
2,25559	-23,087	2,25562
2,25562	-23,087	2,25563
2,25563	-23,087	2,25563

Fue fácil darse cuenta donde está el mínimo real viendo la gráfica, pero el problema estará vigente cuando no sea fácil generar el gráfico o peor aún, cuando no sea una sola variable independiente f(x) sino varias, como funciones del tipo f(a,b,c,d,e)

En la figura, hay dos entradas: A y B, y una salida: C, todo eso son constantes porque son los datos de entrenamiento, no tenemos control sobre estos. Lo que si podemos variar, son los controles análogos. Si queremos saber que tanto debe ajustar cada control análogo, el procedimiento matemático de obtener mínimos, se enfoca solamente en esos controles.



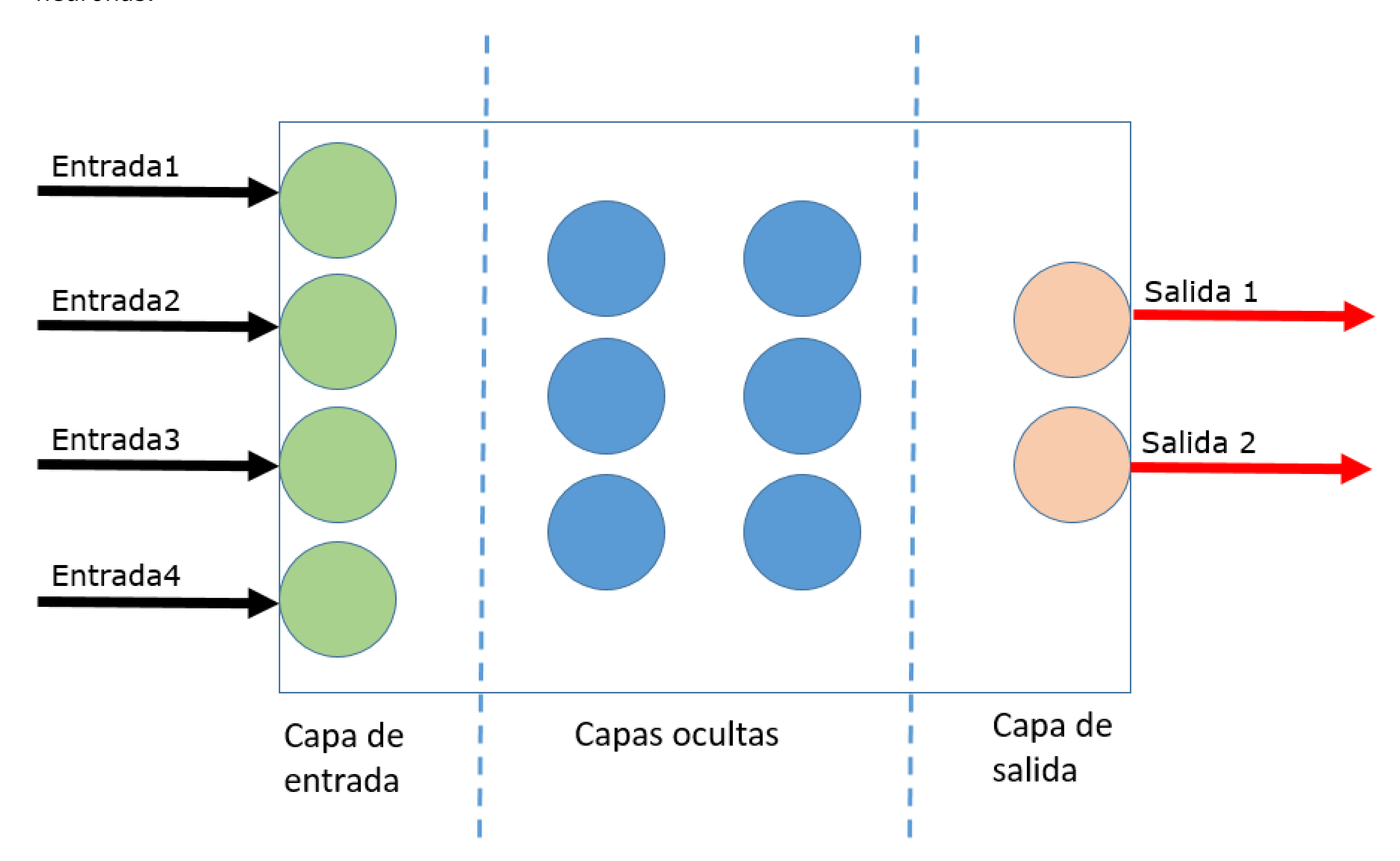
En la figura se aprecian 7 controles o variables: a,b,c,d,e,g,h. ¿Cómo obtener un mínimo? En ese caso se utilizan derivadas parciales, es decir, se deriva por ‘a’ dejando el resto como constantes, luego por ‘b’ dejando el resto constantes y así sucesivamente. Esos mínimos servirán para ir ajustando los controles.

Perceptrón Multicapa

Es un tipo de red neuronal en donde hay varias capas:

1. Capa de entrada
2. Capas ocultas
3. Capa de salida

En la siguiente figura se muestra un ejemplo de perceptrón multicapa, los círculos representan las neuronas. Tiene dos capas ocultas. La capa de entrada con 4 neuronas, las capas ocultas donde cada una tiene 3 neuronas y la capa de salida con 2 neuronas.



Las capas se denotarán con la letra 'N', luego

N_1=4 (capa 1, que es la de entrada, tiene 4 neuronas)

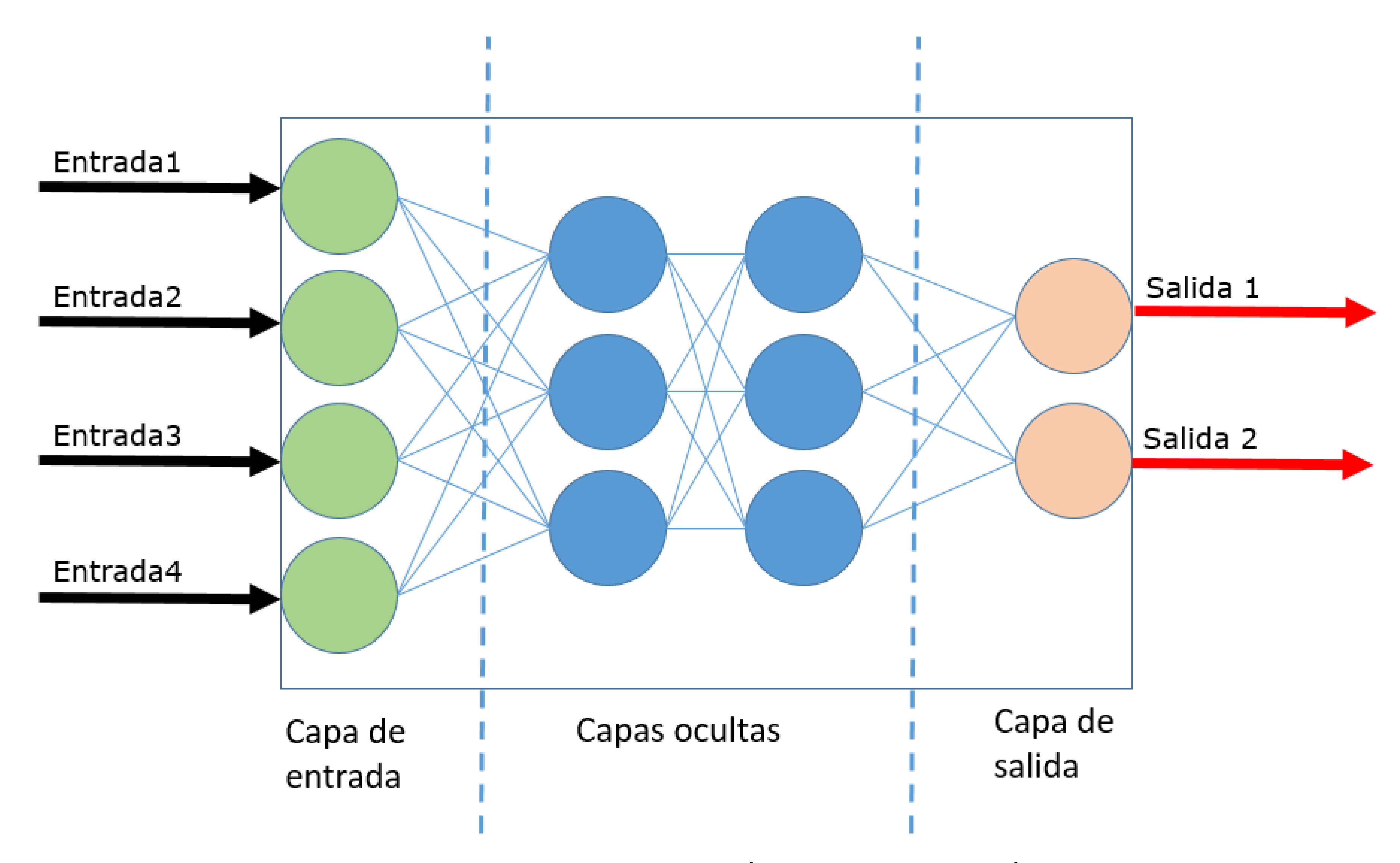
N_2=3 (capa 2, que es oculta, tiene 3 neuronas)

N_3=3 (capa 3, que es oculta, tiene 3 neuronas)

N_4=2 (capa 4, que es la de salida, tiene 2 neuronas)

Las conexiones entre capas del perceptrón multicapa

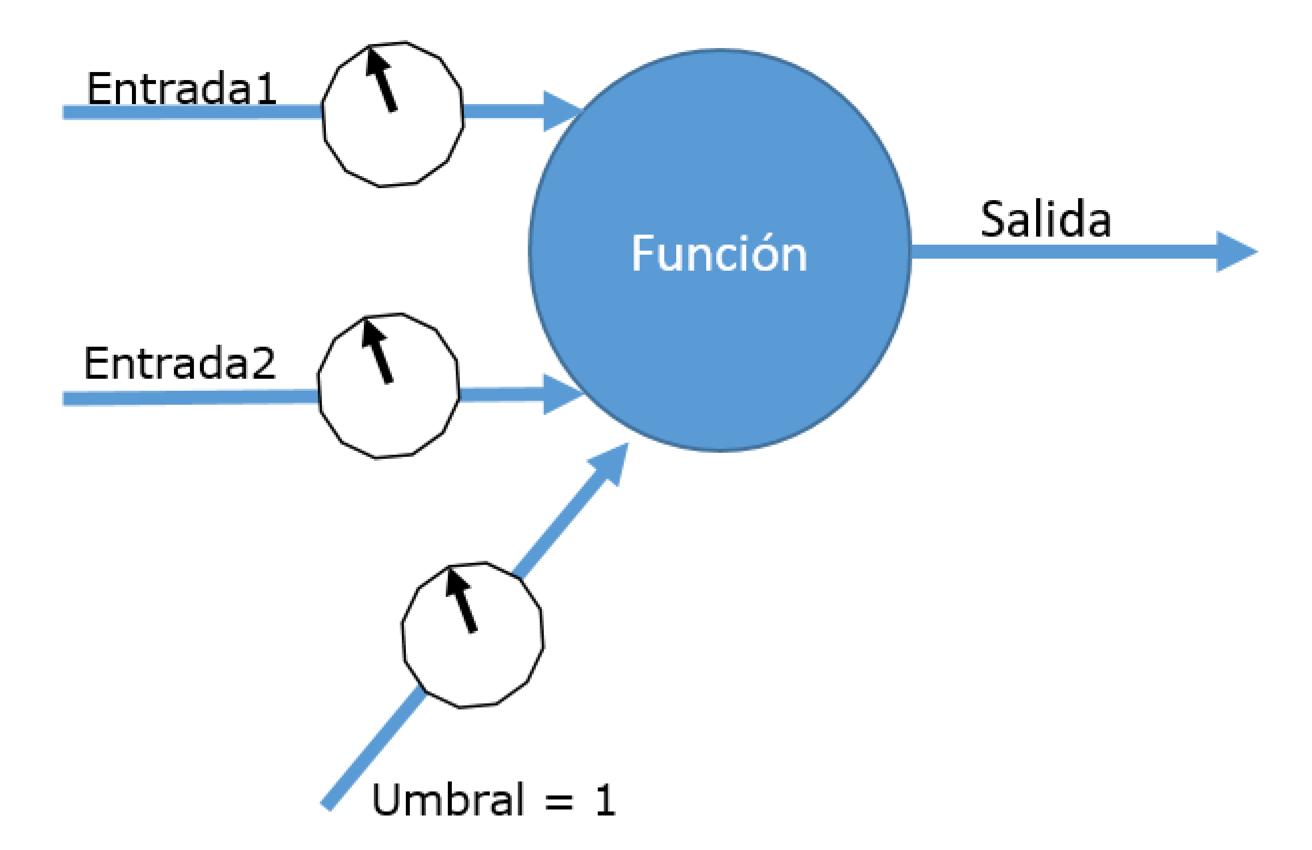
En el perceptrón multicapa, las neuronas de la capa 1 se conectan con las neuronas de la capa 2, las neuronas de la capa 2 con las neuronas de la capa 3 y así sucesivamente. No está permitido conectar neuronas de la capa 1 con las neuronas de la capa 4 por ejemplo, ese salto podrá suceder en otro tipo de redes neuronales pero no en el perceptrón multicapa.



En la capa de entrada no hay procesamiento de la información, tan solo la recepción de los valores de entrada.

Las neuronas

De nuevo se muestra un esquema de cómo es una neurona con dos entradas externas y su salida.



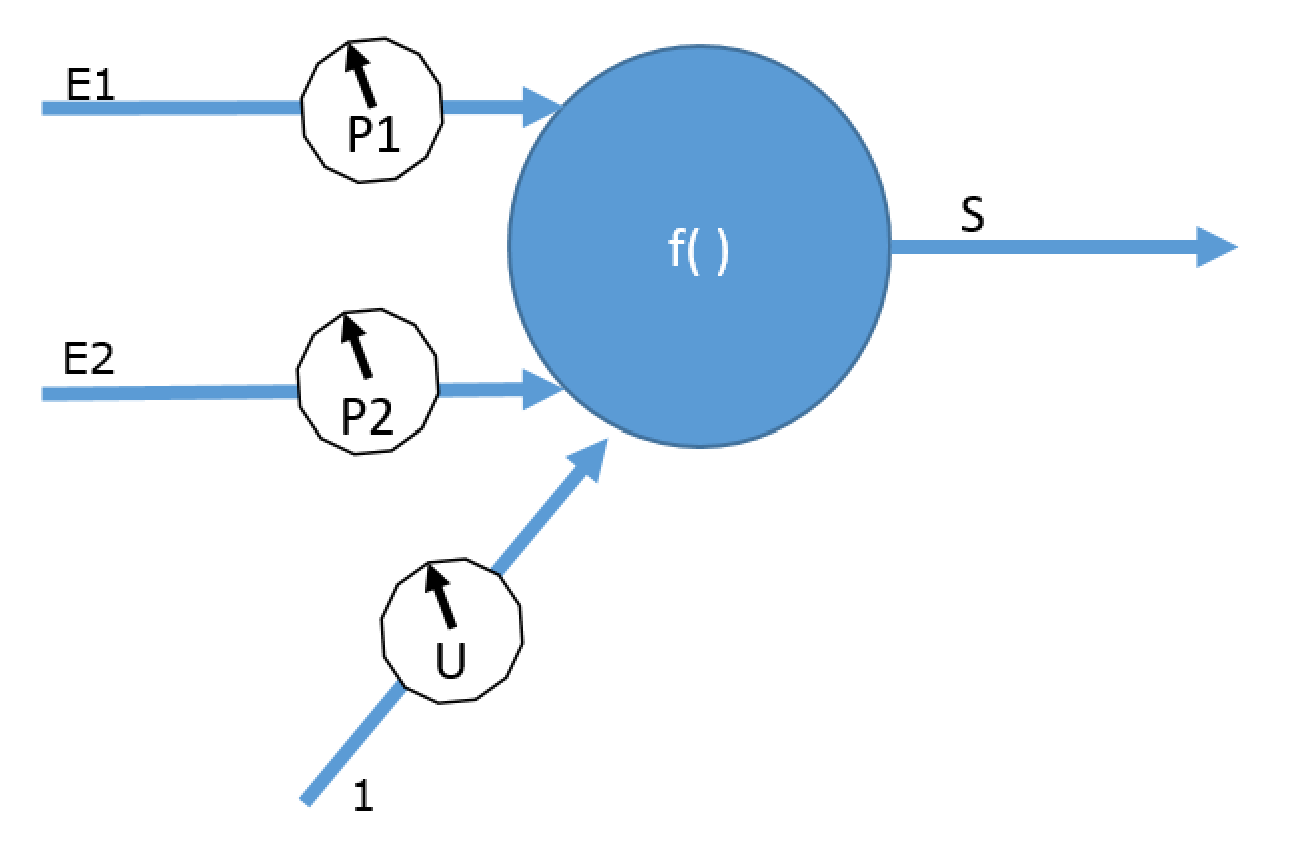
Mostrado como una clase en C#, esta sería su implementación:

```
namespace RedesNeuronales {
    class Neurona {
        public double calculaSalida(double E1, double E2)
        {
            double S;
            //Se hace una operación aquí

            return S;
        }
    }

    class Program {
        static void Main(string[] args)
        {
            Neurona algunaCapasOcultas = new Neurona();
            Neurona algunaCapaSalida = new Neurona();
        }
    }
}
```

En cada entrada hay un peso P1 y P2. Para la entrada interna, que siempre es 1, el peso se llama U



```
namespace RedesNeuronales {
    class Neurona {
        //Pesos para cada entrada P1 y P2; y el peso de la entrada interna U
        private double P1;
        private double P2;
        private double U;

        public double calculaSalida(double E1, double E2)
        {
            double S;

            //Se hace una operación aquí

            return S;
        }
    }

    class Program {
        static void Main(string[] args)
        {
            Neurona algunaCapasOcultas = new Neurona();
            Neurona algunaCapaSalida = new Neurona();
        }
    }
}
```

Ese tipo de neurona está en las capas ocultas y capa de salida del perceptrón multicapa.

Los pesos se inicializan con un valor al azar y un buen sitio es hacerlo en el constructor. En el ejemplo se hace uso de la clase Random y luego NextDouble() que retorna un número real al azar entre 0 y 1.

```
namespace RedesNeuronales {
    class Neurona {
        //Pesos para cada entrada P1 y P2; y el peso de la entrada interna U
        private double P1;
        private double P2;
        private double U;

        public Neurona() //Constructor
        {
            Random azar = new Random();
            P1 = azar.NextDouble();
            P2 = azar.NextDouble();
            U = azar.NextDouble();
        }

        public double calculaSalida(double E1, double E2)
        {
            double S;

            //Se hace una operación aquí

            return S;
        }
    }

    class Program {
        static void Main(string[] args)
        {
            Neurona algunaCapasOcultas = new Neurona();
            Neurona algunaCapaSalida = new Neurona();
        }
    }
}
```

Hay que tener especial cuidado con los generadores de números aleatorios, no es bueno crearlos constantemente porque se corre el riesgo que inicien con una misma semilla (el reloj de la máquina) generando la misma colección de números aleatorios. A continuación se modifica un poco el código para tener un solo generador de números aleatorios y evitar el riesgo de repetir números.

```
namespace RedesNeuronales {
    class Neurona {
        //Pesos para cada entrada P1 y P2; y el peso de la entrada interna U
        private double P1;
        private double P2;
        private double U;

        public Neurona(Random azar) //Constructor
        {
            P1 = azar.NextDouble();
            P2 = azar.NextDouble();
            U = azar.NextDouble();
        }

        public double calculaSalida(double E1, double E2)
        {
            double S;

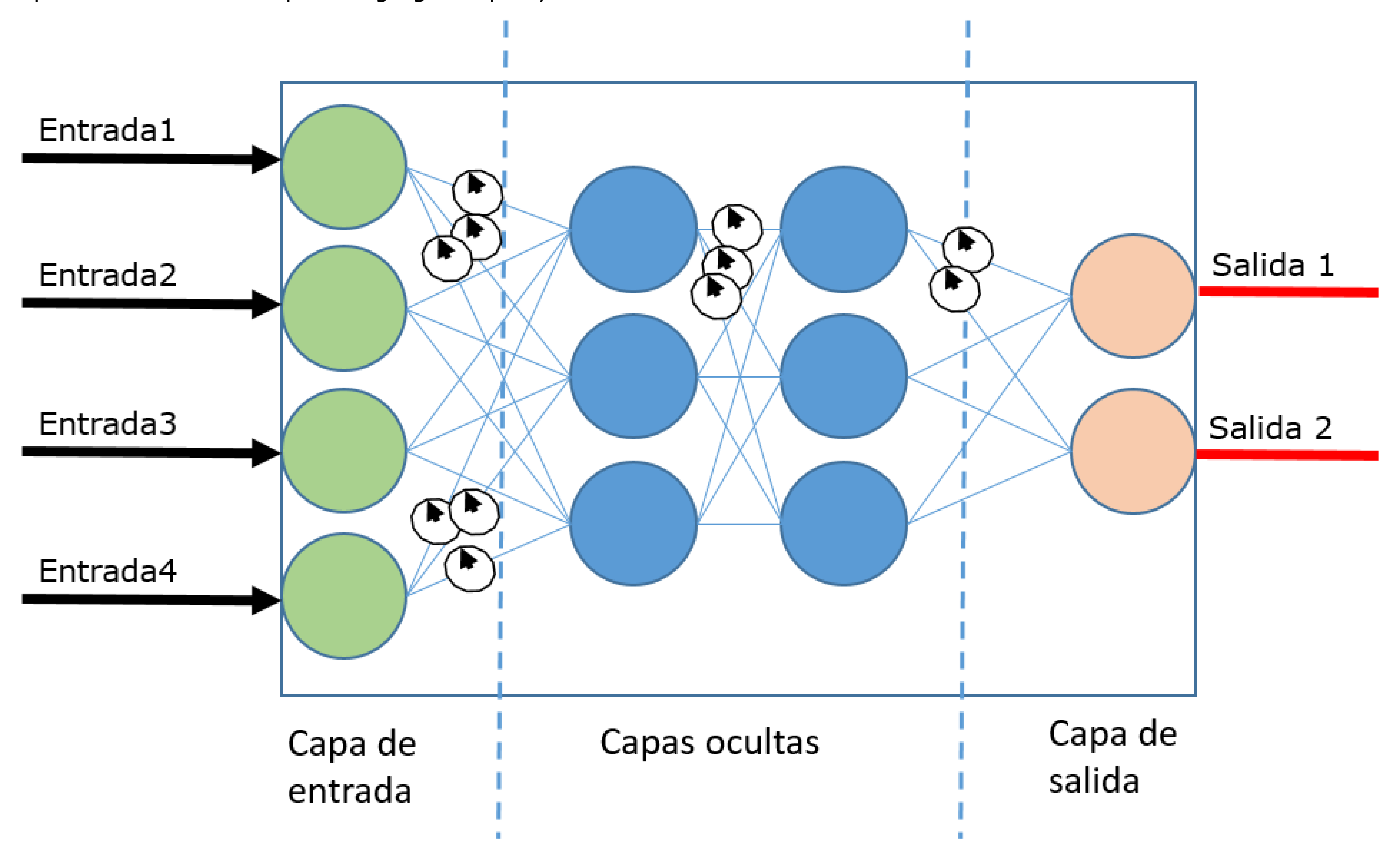
            //Se hace una operación aquí

            return S;
        }
    }

    class Program {
        static void Main(string[] args)
        {
            Random azar = new Random(); //Un solo generador
            Neurona algunaCapasOcultas = new Neurona(azar);
            Neurona algunaCapaSalida = new Neurona(azar);
        }
    }
}
```

Pesos y como nombrarlos

En el gráfico se dibujan algunos pesos (controles analógicos) y como se podrá dilucidar, el número de estos pesos crece rápidamente a medida que se agregan capas y neuronas.



Un ejemplo: Capa 1 tiene 5 neuronas, capa 2 tiene 4 neuronas, luego el total de conexiones entre Capa 1 y Capa 2 son 5*4=20 conexiones, luego son 20 pesos. Nos quedaríamos rápidamente sin letras al nombrar cada peso. Por tal motivo, hay otra forma de nombrarlos y es el siguiente

$$W_{neurona\ inicial, neurona\ final}^{(capa\ de\ donde\ sale\ la\ conexión)}$$

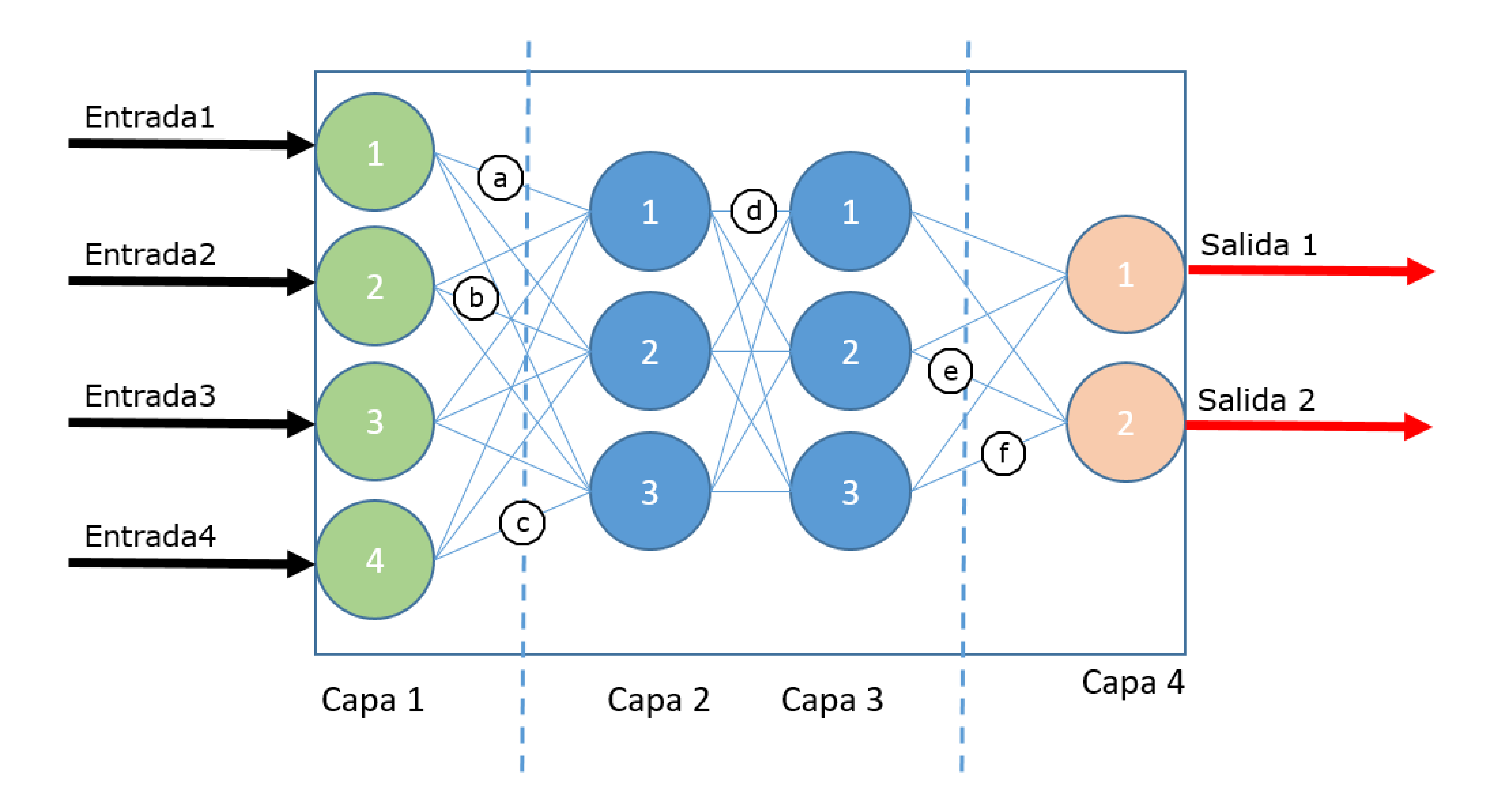
W es la letra inicial de la palabra peso en inglés: Weight

(Capa de donde sale la conexión) Las capas se enumeran desde 1 que sería en este caso la capa de entrada

Neurona inicial, de donde parte la conexión

Neurona final, a donde llega la conexión

A continuación, se muestra el esquema con cada capa y cada neurona con un número



Para nombrar el peso mostrado con la letra 'a' sería entonces

$$w_{neurona\ inicial, neurona\ final}^{(capa\ de\ donde\ sale\ la\ conexión)}$$

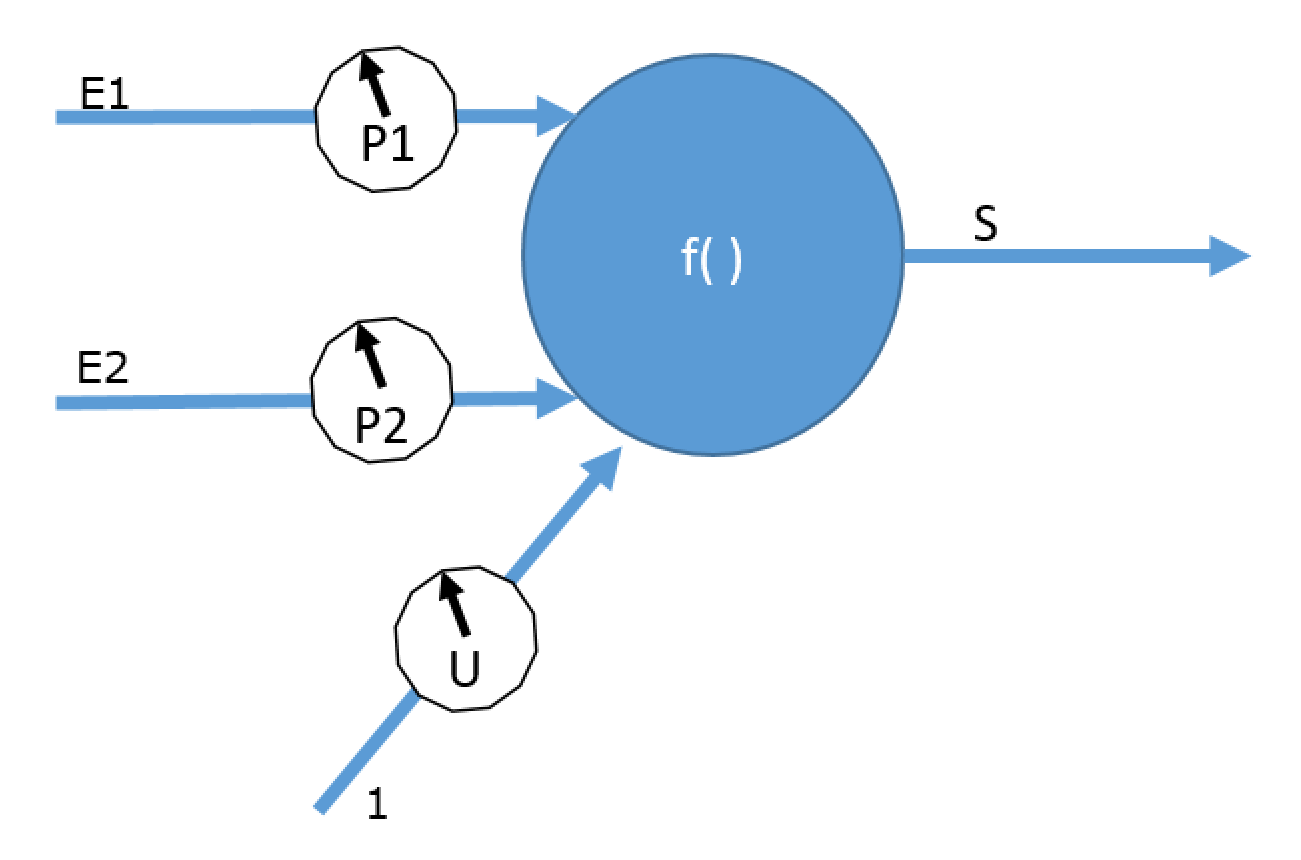
$$w_{1,1}^{(1)}$$

En esta tabla se muestra como se nombrarían los pesos que se han puesto en la gráfica

Peso	Se nombra
a	$w_{1,1}^{(1)}$
b	$w_{2,2}^{(1)}$
c	$w_{4,3}^{(1)}$
d	$w_{1,1}^{(2)}$
e	$w_{2,2}^{(3)}$
f	$w_{3,2}^{(3)}$

La función de activación de la neurona

Viendo de nuevo el esquema de una neurona con dos entradas, una salida y un umbral



La salida se calcula así:

S = f (E1 * P1 + E2 * P2 + 1 * U)

¿Y que es f( )? Es la función de activación. Al principio de este libro se documentó así:

```
Función f(valor)
Inicio
     Si valor > 0 entonces
             retorne 1
     de lo contrario
             retorne 0
     fin si
Fin
```

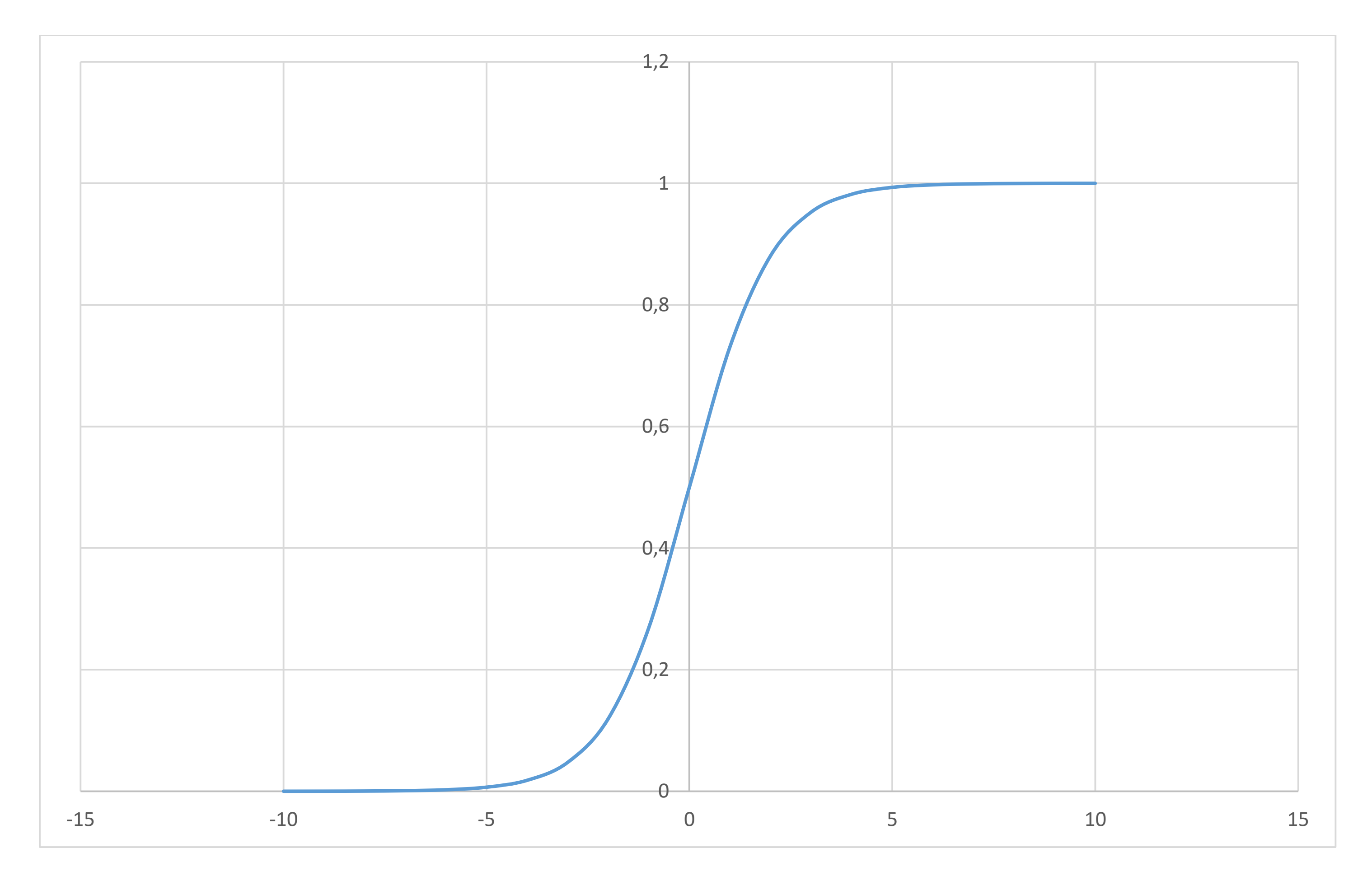
En otros problemas, por lo general, esa función es la sigmoide que tiene la siguiente ecuación:

$$y = \frac{1}{1 + e^{-x}}$$

Esta sería una tabla de valores generados con esa función

x	y
-10	4,5E-05
-9	0,00012
-8	0,00034
-7	0,00091
-6	0,00247
-5	0,00669
-4	0,01799
-3	0,04743
-2	0,1192
-1	0,26894
0	0,5
1	0,73106
2	0,8808
3	0,95257
4	0,98201
5	0,99331
6	0,99753
7	0,99909
8	0,99966
9	0,99988
10	0,99995

Y esta sería su gráfica



Al moverse a la izquierda el valor que toma es 0 y al moverse a la derecha toma el valor de 1. Hay una transición pronunciada de 0 a 1 en el rango [-5 y 5].

¿Qué tiene de especial esta función sigmoide? Su derivada.

Ecuación original:

$$y = \frac{1}{1 + e^{-x}}$$

Derivada de esa ecuación:

$$y' = \frac{e^{-x}}{(1 + e^{-x})^2}$$

Que equivale a esto:

$$y' = y * (1 - y)$$

Demostración de la equivalencia:

$$y' = \frac{1}{1 + e^{-x}} * (1 - \frac{1}{1 + e^{-x}})$$

$$y' = \frac{1}{1 + e^{-x}} - \frac{1}{1 + e^{-x}} * \frac{1}{1 + e^{-x}}$$

$$y' = \frac{1}{1 + e^{-x}} - \frac{1}{(1 + e^{-x})^2}$$

$$y' = \frac{1 + e^{-x} - 1}{(1 + e^{-x})^2}$$

$$y' = \frac{e^{-x}}{(1 + e^{-x})^2}$$

El código del perceptrón multicapa progresa así:

```
using System;

namespace RedesNeuronales {
    class Neurona {
        //Pesos para cada entrada P1 y P2; y el peso de la entrada interna U
        private double P1;
        private double P2;
        private double U;

        public Neurona(Random azar){
            P1 = azar.NextDouble();
            P2 = azar.NextDouble();
            U = azar.NextDouble();
        }

        public double calculaSalida(double E1, double E2) {
            double valor, S;

            valor = E1 * P1 + E2 * P2 + 1 * U;
            S = 1 / (1 + Math.Exp(-valor));

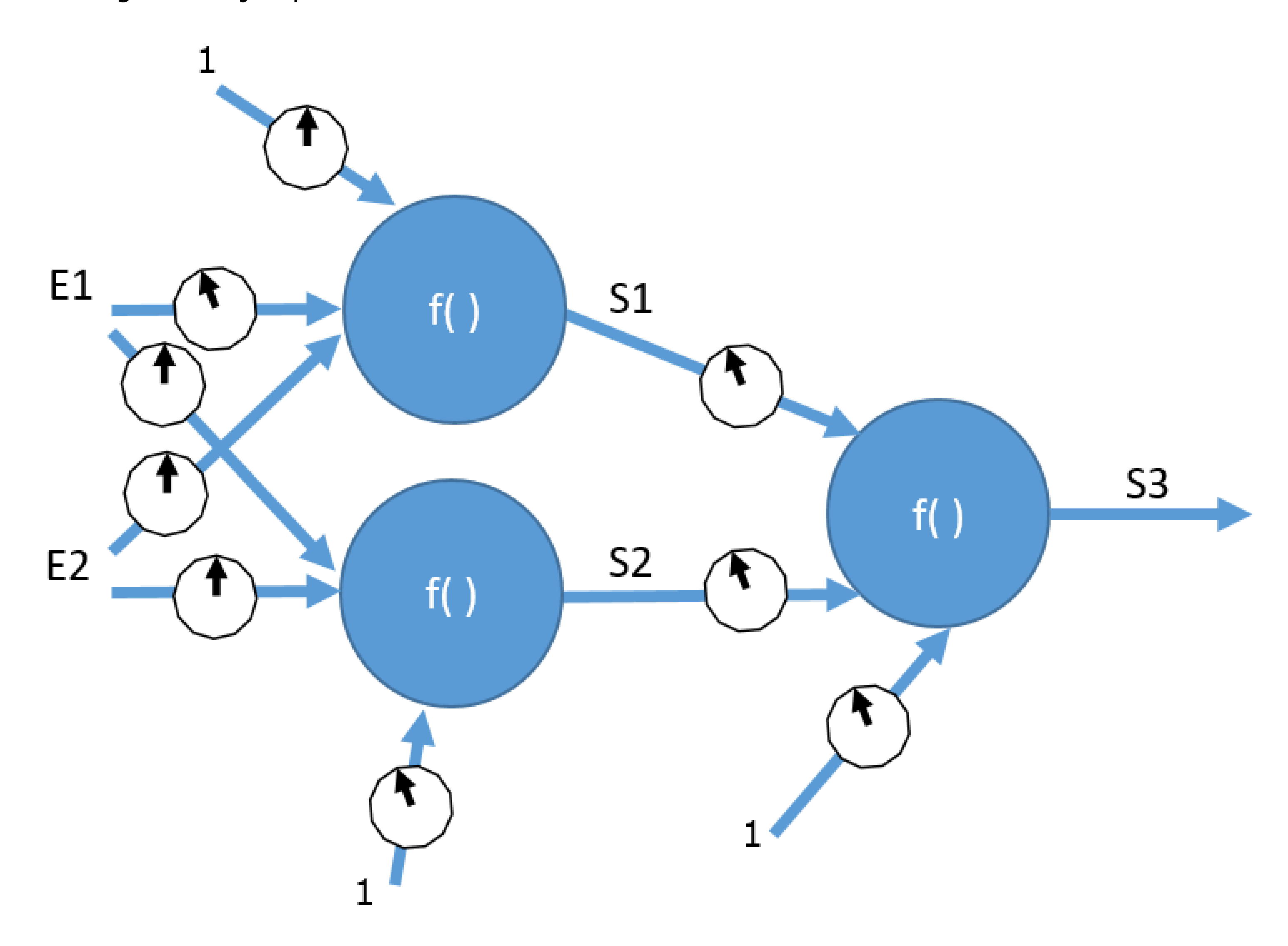
            return S;
        }
    }

    class Program {
        static void Main(string[] args)
        {
            Random azar = new Random(); //Un solo generador de números al azar
            Neurona algunaCapasOcultas = new Neurona(azar);
            Neurona algunaCapaSalida = new Neurona(azar);
        }
    }
}
```

El método calculaSalida implementa el procesamiento de la neurona. Tiene como parámetros las entradas, en este caso, dos entradas E1 y E2. En el interior cada entrada se multiplica con su peso respectivo, se suman, incluyendo la entrada interna (umbral). Una vez con ese valor, se calcula la salida con la sigmoide.

Introducción al algoritmo de propagación hacia atrás de errores

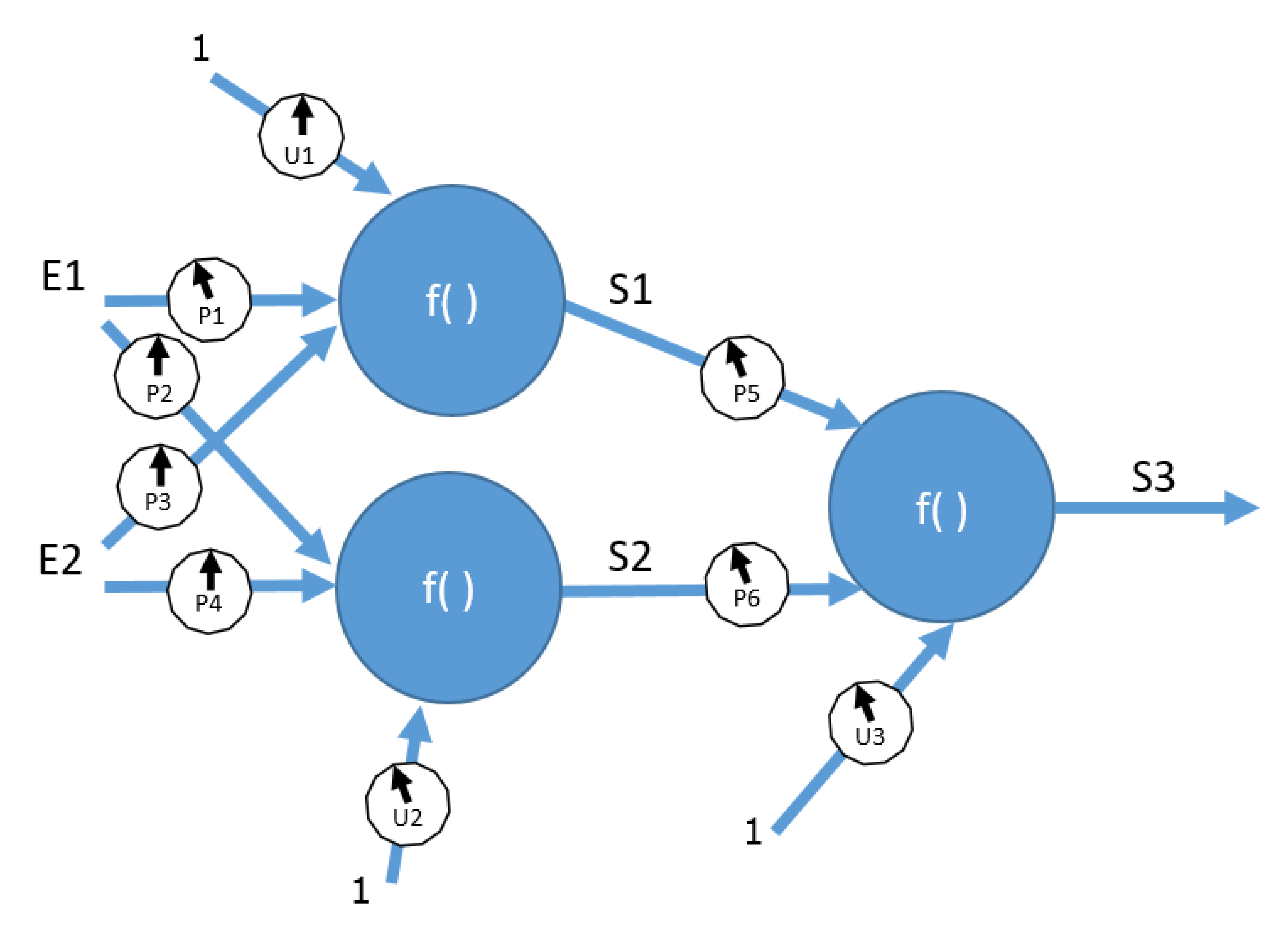
En el siguiente ejemplo vemos una conexión entre tres neuronas



E1 y E2 son las entradas externas, los valores que da el problema. Se observa que E1 entra con un peso en la neurona de arriba y con otro peso en la neurona de abajo. Sucede lo mismo con la entrada E2. Tanto la neurona de arriba como la de abajo tienen sus propias entradas internas.

Lo interesante viene después, porque la salida de la neurona de arriba que es S1 y la salida de la neurona de abajo que es S2 se convierten en entradas para la neurona de la derecha y esas entradas a su vez tienen sus propios pesos. Al final el sistema genera una salida S3 que es la respuesta final de la red neuronal.

¿Qué importancia tiene eso? Que si queremos ajustar S3 al resultado que esperamos, entonces retrocedemos a las entradas (S1 y S2) de esa neurona de la derecha ajustando sus pesos respectivos y por supuesto, ese ajuste nos hace retroceder más aún hasta mirar los pesos de las neuronas de arriba y abajo. Eso se conocerá como el algoritmo de propagación hacia atrás de errores o retro propagación (backpropagation).



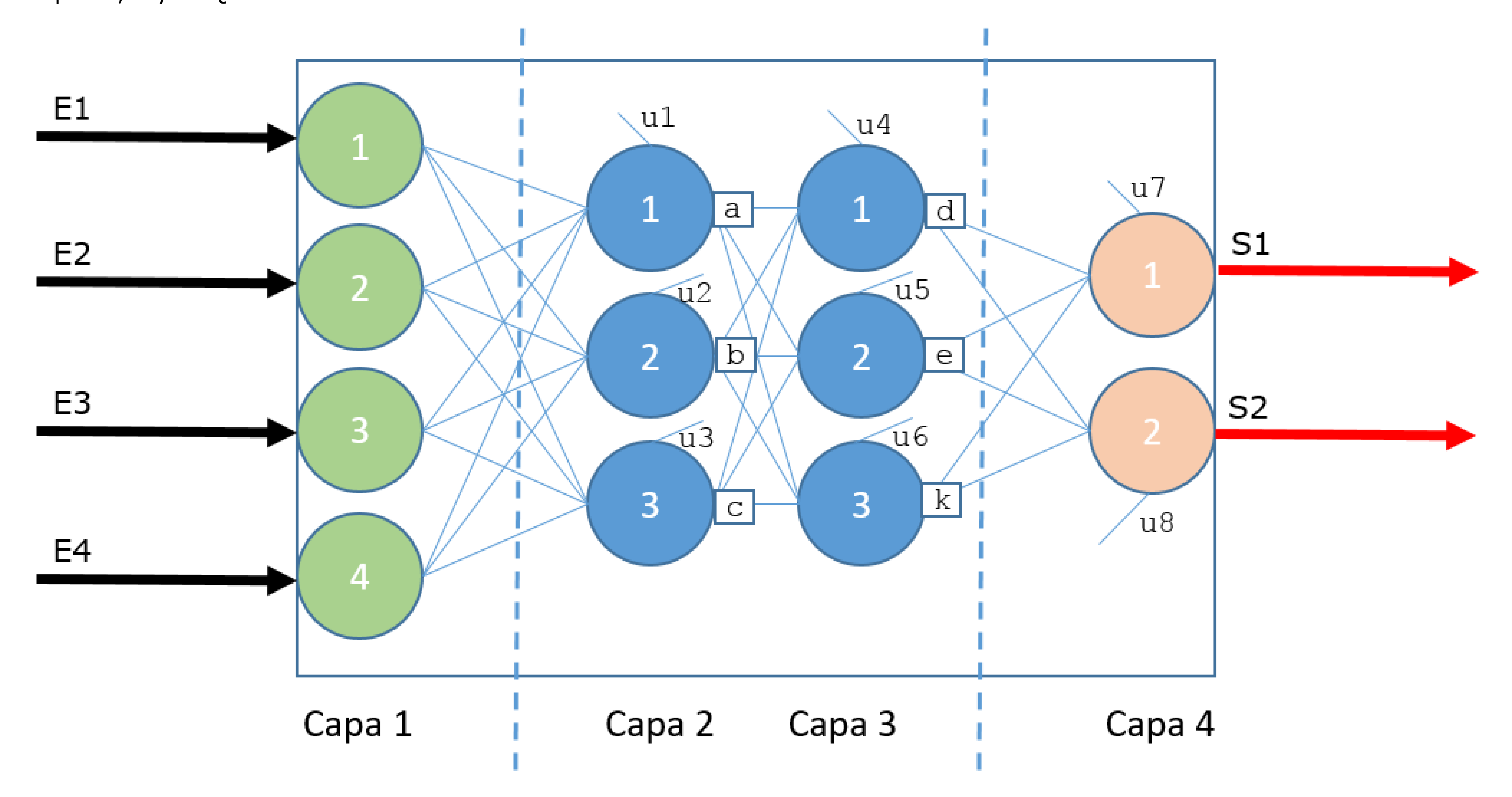
Luego:

$$S1 = f(E1 * P1 + E2 * P3 + 1 * U1)$$
$$S2 = f(E1 * P2 + E2 * P4 + 1 * U2)$$
$$S3 = f(S1 * P5 + S2 * P6 + 1 * U3)$$

Se concluye entonces que

$$S3 = f(f(E1 * P1 + E2 * P3 + 1 * U1) * P5 + f(E1 * P2 + E2 * P4 + 1 * U2) * P6 + 1 * U3)$$

Volviendo al perceptrón multicapa, le ponemos nombre a las salidas de las neuronas que hacen procesamiento, es decir, las capas 2, 3 y 4. Quedando así:



Y recordando la forma de nombrar los pesos

$$w_{neurona\ inicial, neurona\ final}^{(capa\ de\ donde\ sale\ la\ conexión)}$$

Luego

$$S1 = f\left(d * w_{1,1}^{(3)} + e * w_{2,1}^{(3)} + k * w_{3,1}^{(3)} + 1 * u7\right)$$

$$S2 = f\left(d * w_{1,2}^{(3)} + e * w_{2,2}^{(3)} + k * w_{3,2}^{(3)} + 1 * u8\right)$$

$$d = f\left(a * w_{1,1}^{(2)} + b * w_{2,1}^{(2)} + c * w_{3,1}^{(2)} + 1 * u4\right)$$

$$e = f\left(a * w_{1,2}^{(2)} + b * w_{2,2}^{(2)} + c * w_{3,2}^{(2)} + 1 * u5\right)$$

$$k = f\left(a * w_{1,3}^{(2)} + b * w_{2,3}^{(2)} + c * w_{3,3}^{(2)} + 1 * u6\right)$$

$$a = f\left(E1 * w_{1,1}^{(1)} + E2 * w_{2,1}^{(1)} + E3 * w_{3,1}^{(1)} + E4 * w_{4,1}^{(1)} + 1 * u1\right)$$

$$b = f\left(E1 * w_{1,2}^{(1)} + E2 * w_{2,2}^{(1)} + E3 * w_{3,2}^{(1)} + E4 * w_{4,2}^{(1)} + 1 * u2\right)$$

$$c = f\left(E1 * w_{1,3}^{(1)} + E2 * w_{2,3}^{(1)} + E3 * w_{3,3}^{(1)} + E4 * w_{4,3}^{(1)} + 1 * u3\right)$$

Nombrando las salidas, los umbrales y las capas

En el gráfico anterior se aprecia que las salidas han sido nombradas a, b, c, d, e, k y los umbrales u1, u2, u3, u4, u5, u6, u7, u8. Esa no es la mejor manera de nombrarlos porque un perceptrón multicapa podría tener una buena cantidad de neuronas a tal punto que nos quedaríamos sin letras y por otro lado habría que mirar constantemente el gráfico para dar con la salida o umbral nombrado. Por esa razón, hay una mejor manera de nombrarlos.

Salidas

$$a_{neurona\ de\ esa\ salida}^{(capa\ de\ la\ neurona\ de\ esa\ salida)}$$

Umbrales

$$u_{neurona\ que\ tiene\ esa\ entrada\ interna}^{(capa\ de\ la\ neurona\ que\ tiene\ esa\ entrada\ interna)}$$

Capas

$$n_{número\ de\ la\ capa}$$

Volviendo al gráfico, entonces:

Como se nombró antes	Nueva nomenclatura
a	$a_1^{(2)}$
b	$a_2^{(2)}$
c	$a_3^{(2)}$
d	$a_1^{(3)}$
e	$a_2^{(3)}$
k	$a_3^{(3)}$
u1	$u_1^{(2)}$
u2	$u_2^{(2)}$
u3	$u_3^{(2)}$
u4	$u_1^{(3)}$
u5	$u_2^{(3)}$
u6	$u_3^{(3)}$

u7	$u_1^{(4)}$
u8	$u_2^{(4)}$
Capa 1	n_1
Capa 2	n_2
Capa 3	n_3
Capa 4	n_4

El gráfico anterior las capas tienen valores que es el número de neuronas en cada capa, luego

$$n_1 = 4$$
$$n_2 = 3$$
$$n_3 = 3$$
$$n_4 = 2$$

Las ecuaciones cambian así:

$$S1 = f\left(a_1^{(3)} * w_{1,1}^{(3)} + a_2^{(3)} * w_{2,1}^{(3)} + a_3^{(3)} * w_{3,1}^{(3)} + 1 * u_1^{(4)}\right)$$
$$S2 = f\left(a_1^{(3)} * w_{1,2}^{(3)} + a_2^{(3)} * w_{2,2}^{(3)} + a_3^{(3)} * w_{3,2}^{(3)} + 1 * u_2^{(4)}\right)$$
$$a_1^{(3)} = f\left(a_1^{(2)} * w_{1,1}^{(2)} + a_2^{(2)} * w_{2,1}^{(2)} + a_3^{(2)} * w_{3,1}^{(2)} + 1 * u_1^{(3)}\right)$$
$$a_2^{(3)} = f\left(a_1^{(2)} * w_{1,2}^{(2)} + a_2^{(2)} * w_{2,2}^{(2)} + a_3^{(2)} * w_{3,2}^{(2)} + 1 * u_2^{(3)}\right)$$
$$a_3^{(3)} = f\left(a_1^{(2)} * w_{1,3}^{(2)} + a_2^{(2)} * w_{2,3}^{(2)} + a_3^{(2)} * w_{3,3}^{(2)} + 1 * u_2^{(3)}\right)$$
$$a_1^{(2)} = f\left(E1 * w_{1,1}^{(1)} + E2 * w_{2,1}^{(1)} + E3 * w_{3,1}^{(1)} + E4 * w_{4,1}^{(1)} + 1 * u_1^{(2)}\right)$$
$$a_2^{(2)} = f\left(E1 * w_{1,2}^{(1)} + E2 * w_{2,2}^{(1)} + E3 * w_{3,2}^{(1)} + E4 * w_{4,2}^{(1)} + 1 * u_2^{(2)}\right)$$
$$a_3^{(2)} = f\left(E1 * w_{1,3}^{(1)} + E2 * w_{2,3}^{(1)} + E3 * w_{3,3}^{(1)} + E4 * w_{4,3}^{(1)} + 1 * u_3^{(2)}\right)$$

¿Qué hay de E1, E2, E3 y E4? Podrían tomarse como salidas de las neuronas de la capa 1. Cabe recordar que en esa capa 1 no hay procesamiento, luego los datos que entran son los mismos que salen. Luego:

Como se nombró antes	Nueva nomenclatura
E1	$a_1^{(1)}$
E2	$a_2^{(1)}$
E3	$a_3^{(1)}$
E4	$a_4^{(1)}$

Luego las tres últimas ecuaciones quedan así:

$$a_1^{(2)} = f\left(a_1^{(1)} * w_{1,1}^{(1)} + a_2^{(1)} * w_{2,1}^{(1)} + a_3^{(1)} * w_{3,1}^{(1)} + a_4^{(1)} * w_{4,1}^{(1)} + 1 * u_1^{(2)}\right)$$

$$a_2^{(2)} = f\left(a_1^{(1)} * w_{1,2}^{(1)} + a_2^{(1)} * w_{2,2}^{(1)} + a_3^{(1)} * w_{3,2}^{(1)} + a_4^{(1)} * w_{4,2}^{(1)} + 1 * u_2^{(2)}\right)$$

$$a_3^{(2)} = f\left(a_1^{(1)} * w_{1,3}^{(1)} + a_2^{(1)} * w_{2,3}^{(1)} + a_3^{(1)} * w_{3,3}^{(1)} + a_4^{(1)} * w_{4,3}^{(1)} + 1 * u_3^{(2)}\right)$$

Y generalizando se puede decir que

$$a_i^{(k)} = f\left(a_j^{(k-1)} * w_{j,i}^{(k-1)} + a_{j+1}^{(k-1)} * w_{j+1,i}^{(k-1)} + a_{j+2}^{(k-1)} * w_{j+2,i}^{(k-1)} + a_{j+3}^{(k-1)} * w_{j+3,i}^{(k-1)} + 1 * u_i^{(k)}\right)$$

$$a_i^{(k)} = f(\, u_i^{(k)} + \sum_{j=1}^{n_{k-1}} a_j^{(k-1)} * w_{j,i}^{(k-1)} \,)$$

Regla de la cadena

Para continuar con el algoritmo de propagación hacia atrás, cabe recordar esta regla matemática llamada regla de la cadena

$$[f(g(x))]' = f'(g(x)) * g'(x)$$

Un ejemplo:

$$g(x) = 3 * x^2$$

$$f(p) = 7 - p^3$$

Luego

$$f(g(x)) = 7 - (3 * x^2)^3$$

Derivando

$$[f(g(x))]' = -\frac{2 * 3^3 * 3 * (x^2)^3}{x} = -\frac{162 * x^6}{x} = -162 * x^5$$

Usando la regla de la cadena

$$[f(g(x))]' = f'(g(x)) * g'(x) = (7 - p^3)' * (3 * x^2)' =$$

$$(0 - 3 * p^2) * (6 * x) = (0 - 3 * (3 * x^2)^2) * (6 * x) =$$

$$(0 - 3 * (9 * x^4)) * (6 * x) = -162 * x^5$$

Derivadas parciales

Dada una ecuación que tenga dos o más variables independientes, es posible derivar por una variable considerando las demás constantes, eso es conocido como derivada parcial

Ejemplo de una ecuación con tres variables independientes

$$q = a^2 + b^3 + c^4$$

Su derivada parcial con respecto a la variable **b** sería

$$\frac{\partial q}{\partial b}(a, b, c) = 0 + 3 * b^2 + 0$$

$$\frac{\partial q}{\partial b}(a, b, c) = 3 * b^2$$

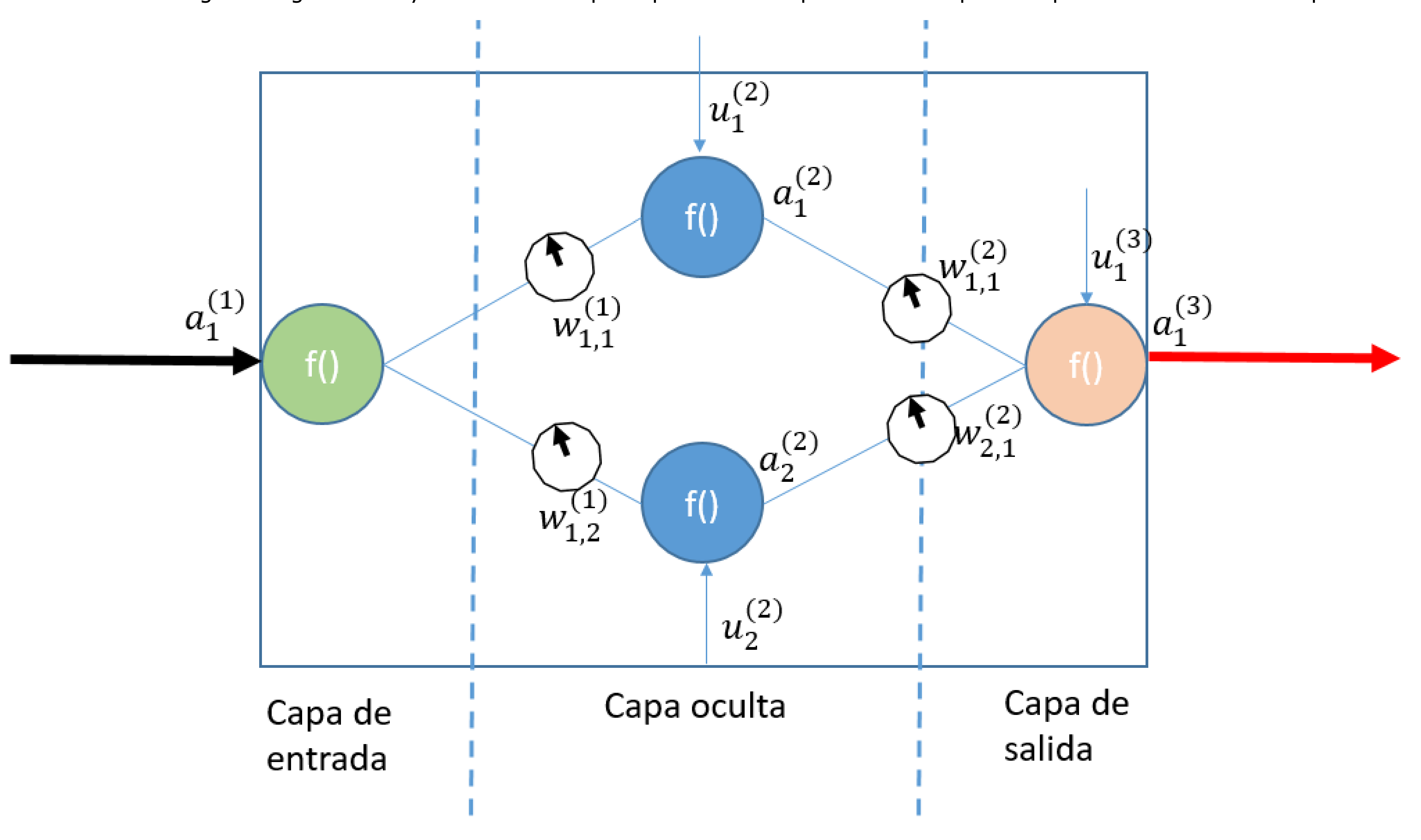
Y con respecto a la variable **a** sería

$$\frac{\partial q}{\partial a}(a, b, c) = 2 * a + 0 + 0$$

$$\frac{\partial q}{\partial a}(a, b, c) = 2 * a$$

Las derivadas en el algoritmo de propagación hacia atrás

Observamos el siguiente gráfico muy sencillo de un perceptrón multicapa. Recordar que la capa de entrada no hace proceso.



$$a_1^{(3)} = f(a_1^{(2)} * w_{1,1}^{(2)} + a_2^{(2)} * w_{2,1}^{(2)} + 1 * u_1^{(3)})$$

$$a_1^{(2)} = f(a_1^{(1)} * w_{1,1}^{(1)} + 1 * u_1^{(2)})$$

$$a_2^{(2)} = f(a_1^{(1)} * w_{1,2}^{(1)} + 1 * u_2^{(2)})$$

Luego reemplazando

$$a_1^{(3)} = f(f(a_1^{(1)} * w_{1,1}^{(1)} + 1 * u_1^{(2)}) * w_{1,1}^{(2)} + f(a_1^{(1)} * w_{1,2}^{(1)} + 1 * u_2^{(2)}) * w_{2,1}^{(2)} + 1 * u_1^{(3)})$$

Simplificando

$$a_1^{(3)} = f(f(a_1^{(1)} * w_{1,1}^{(1)} + u_1^{(2)}) * w_{1,1}^{(2)} + f(a_1^{(1)} * w_{1,2}^{(1)} + u_2^{(2)}) * w_{2,1}^{(2)} + u_1^{(3)})$$

Recordar que la función f( ) es una sigmoidea, por lo tanto al derivar:

$$f(r)' = f(r) * (1 - f(r))$$

¿Qué sucedería si r es a su vez una función?

$$r = g(k)$$

Que si se deriva r aplicando la regla de la cadena tenemos

$$f(r)' = f[g(k)]' = f^{'}[g(k)] * [g(k)]^{'}$$

Y como f( ) es sigmoidea, entonces al derivar

$$f(r)' = f[g(k)]' = f^{'}[g(k)] * [g(k)]^{'} = f(g(k)) * (1 - f(g(k)) * [g(k)]'$$

¡OJO! g(k) es una función sigmoidea y además k es una función polinómica, luego la derivada de [g(k)] aplicando la regla de la cadena sería:

$$[g(k)]' = g'(k) * k' = g(k) * (1 - g(k)) * k'$$

La derivada queda así

$$f(r)' = f[g(k)]' = f^{'}[g(k)] * [g(k)]^{'} = f(g(k)) * (1 - f(g(k)) * g(k) * (1 - g(k)) * k'$$

Simplificando un poco

$$f(r)' = f(r) * \left(1 - f(r)\right) * g(k) * \left(1 - g(k)\right) * k'$$

Esta es la ecuación que se va a derivar parcialmente con respecto a un peso

$$a_1^{(3)} = f(f(a_1^{(1)} * w_{1,1}^{(1)} + u_1^{(2)}) * w_{1,1}^{(2)} + f(a_1^{(1)} * w_{1,2}^{(1)} + u_2^{(2)}) * w_{2,1}^{(2)} + u_1^{(3)})$$

En el ejemplo, se deriva con respecto a $w_{1,1}^{(1)}$ (una derivada parcial). En rojo se pone que ecuación interna es derivable con respecto a $w_{1,1}^{(1)}$

$$\frac{\partial a_1^{(3)}}{\partial w_{1,1}^{(1)}} = [f\left(\left[f\left(a_1^{(1)} * w_{1,1}^{(1)} + u_1^{(2)}\right) * w_{1,1}^{(2)}\right]' + 0 + 0\right)]'$$

Para derivar entonces se deriva la f externa (que está en negro y es f(r)), luego la f interna (que está en rojo y es g(k) y que la multiplica la constante $w_{1,1}^{(2)}$) y por último el polinomio (que es k) que está en verde porque allí está $w_{1,1}^{(1)}$. Hay tres derivaciones.

Sabiendo que:

$$f(r) = a_1^{(3)}$$

Y

$$g(k) = a_1^{(2)} = f(a_1^{(1)} * w_{1,1}^{(1)} + u_1^{(2)})$$

Entonces

$$f(r)' = f(r) * \left(1 - f(r)\right) * g(k) * \left(1 - g(k)\right) * k'$$

$$\frac{\partial a_1^{(3)}}{\partial w_{1,1}^{(1)}} = a_1^{(3)} * \left(1 - a_1^{(3)}\right) * a_1^{(2)} * \left(1 - a_1^{(2)}\right) * w_{1,1}^{(2)} * a_1^{(1)}$$

Para otro peso, en rojo se pone que ecuación interna es derivable con respecto a $w_{1,2}^{(1)}$ sabiendo que:

$$f(r) = a_1^{(3)}$$

Y

$$g(k) = a_2^{(2)} = f(a_1^{(1)} * w_{1,2}^{(1)} + u_2^{(2)})$$

$$\frac{\partial a_1^{(3)}}{\partial w_{1,2}^{(1)}} = [f\left(0 + \left[f\left(a_1^{(1)} * w_{1,2}^{(1)} + u_2^{(2)}\right) * w_{2,1}^{(2)}\right]' + 0\right)]'$$

Y el modelo es:

$$f(r)' = f(r) * \left(1 - f(r)\right) * g(k) * \left(1 - g(k)\right) * k'$$

Luego

$$\frac{\partial a_1^{(3)}}{\partial w_{1,2}^{(1)}} = a_1^{(3)} * \left(1 - a_1^{(3)}\right) * a_2^{(2)} * \left(1 - a_2^{(2)}\right) * w_{2,1}^{(2)} * a_1^{(1)}$$

Generalizando

$$\frac{\partial a_1^{(3)}}{\partial w_{1,j}^{(1)}} = a_1^{(3)} * \left(1 - a_1^{(3)}\right) * a_j^{(2)} * \left(1 - a_j^{(2)}\right) * w_{j,1}^{(2)} * a_1^{(1)}$$

Donde j puede ser 1 o 2. Esa sería la generalización para los pesos $w_{1,1}^{(1)}$ y $w_{1,2}^{(1)}$

¿Qué hay de los pesos $w_{1,1}^{(2)}$ y $w_{2,1}^{(2)}$?

$$a_1^{(3)} = f(f(a_1^{(1)} * w_{1,1}^{(1)} + u_1^{(2)}) * w_{1,1}^{(2)} + f(a_1^{(1)} * w_{1,2}^{(1)} + u_2^{(2)}) * w_{2,1}^{(2)} + u_1^{(3)})$$

$$\frac{\partial a_1^{(3)}}{\partial w_{1,1}^{(2)}} = [f\left(\left[f\left(a_1^{(1)} * w_{1,1}^{(1)} + u_1^{(2)}\right) * w_{1,1}^{(2)}\right]' + 0 + 0\right)]'$$

Observamos que es $w_{1,1}^{(2)}$ con la que se deriva, luego:

$$\frac{\partial a_1^{(3)}}{\partial w_{1,1}^{(2)}} = [f\left(f\left(a_1^{(1)} * w_{1,1}^{(1)} + u_1^{(2)}\right) + 0 + 0\right)]'$$

Y como

$$a_1^{(2)} = f(a_1^{(1)} * w_{1,1}^{(1)} + u_1^{(2)})$$

Y el modelo es

$$f(r)' = f(r) * (1 - f(r)) * g(k) * (1 - g(k)) * k'$$

entonces

$$\frac{\partial a_1^{(3)}}{\partial w_{1,1}^{(2)}} = a_1^{(3)} * \left(1 - a_1^{(3)}\right) * a_1^{(2)}$$

luego

$$\frac{\partial a_1^{(3)}}{\partial w_{2,1}^{(2)}} = a_1^{(3)} * \left(1 - a_1^{(3)}\right) * a_2^{(2)}$$

Generalizando

$$\frac{\partial a_1^{(3)}}{\partial w_{j,1}^{(2)}} = a_1^{(3)} * \left(1 - a_1^{(3)}\right) * a_j^{(2)}$$

Donde j=1 o 2

Faltan los umbrales

$$a_1^{(3)} = f(f(a_1^{(1)} * w_{1,1}^{(1)} + u_1^{(2)}) * w_{1,1}^{(2)} + f(a_1^{(1)} * w_{1,2}^{(1)} + u_2^{(2)}) * w_{2,1}^{(2)} + u_1^{(3)})$$

$$\frac{\partial a_1^{(3)}}{\partial u_1^{(2)}} = [f\left(\left[f\left(a_1^{(1)} * w_{1,1}^{(1)} + u_1^{(2)}\right) * w_{1,1}^{(2)}\right]' + 0 + 0\right)]'$$

Y como

$$a_1^{(2)} = f(a_1^{(1)} * w_{1,1}^{(1)} + 1 * u_1^{(2)})$$

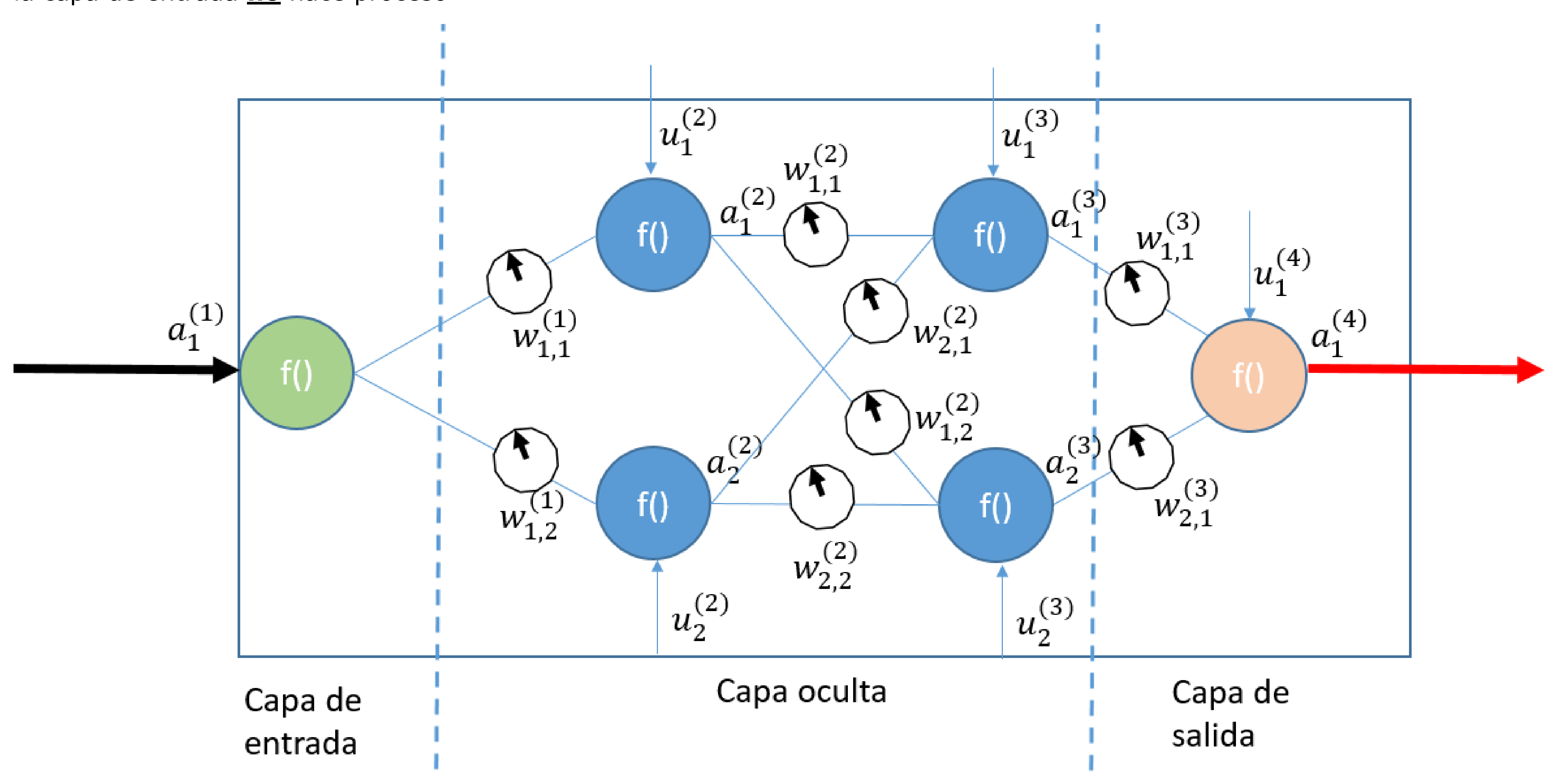
entonces

$$\frac{\partial a_1^{(3)}}{\partial u_1^{(2)}} = a_1^{(3)} * \left(1 - a_1^{(3)}\right) * w_{1,1}^{(2)} * a_1^{(2)} * (1 - a_1^{(2)})$$

en el siguiente umbral

$$\frac{\partial a_1^{(3)}}{\partial u_2^{(2)}} = a_1^{(3)} * \left(1 - a_1^{(3)}\right) * w_{2,1}^{(2)} * a_2^{(2)} * (1 - a_2^{(2)})$$

Con un ejemplo más complejo en el que la capa oculta tiene dos capas de neuronas y cada capa tiene dos neuronas, y ¡OJO! la capa de entrada **no** hace proceso



Se buscan los caminos para $w_{1,1}^{(1)}$, entonces hay dos marcados en rojo

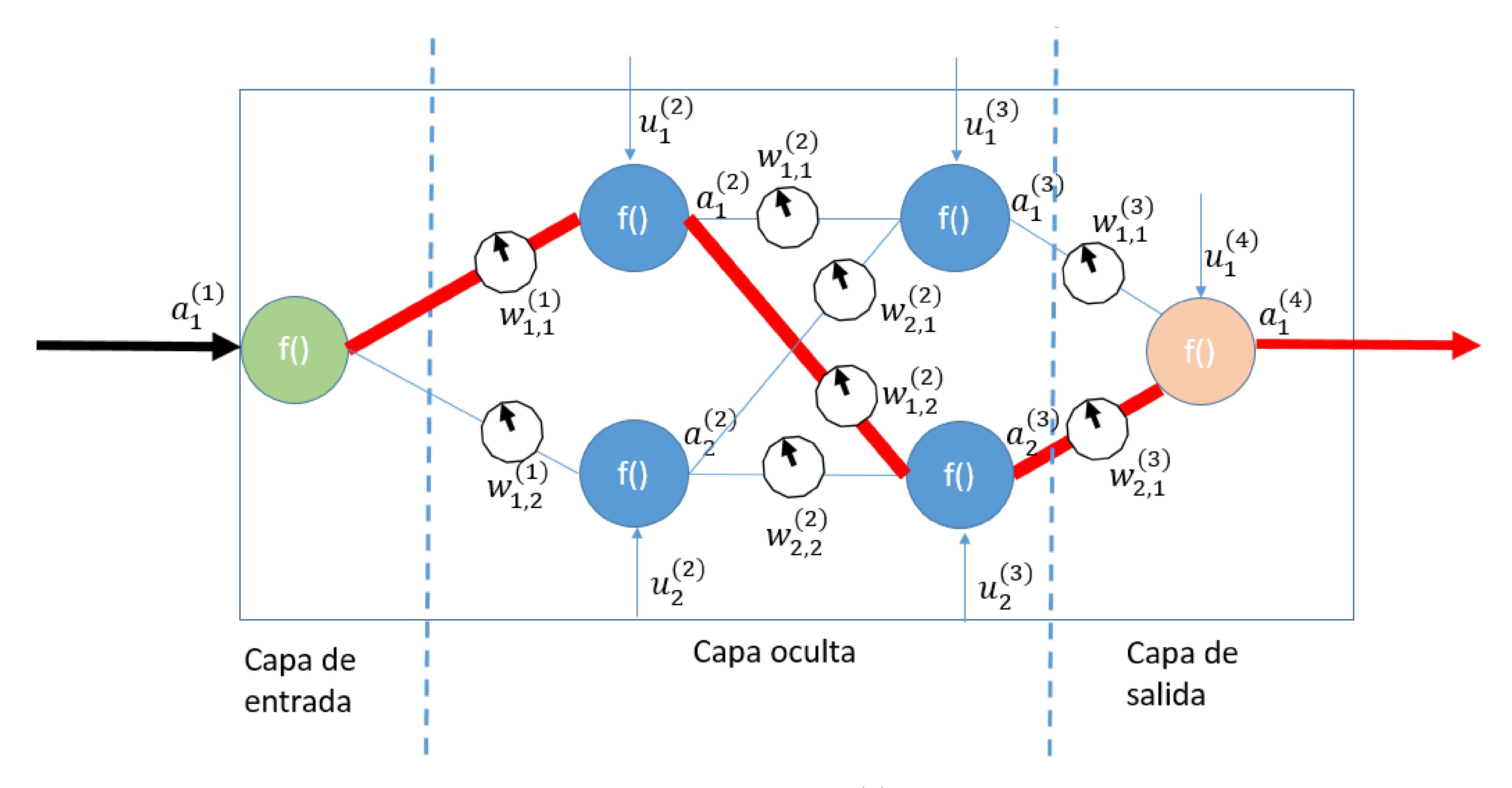
$u_1^{(2)}$ $w_{1,1}^{(2)}$ $u_1^{(3)}$ $a_1^{(2)}$ f() f() $a_1^{(3)}$ $w_{1,1}^{(3)}$ $u_1^{(4)}$ $a_1^{(1)}$ f() $w_{1,1}^{(1)}$ $w_{2,1}^{(2)}$ f() $a_1^{(4)}$ $w_{1,2}^{(2)}$ $a_2^{(2)}$ $a_2^{(3)}$ f() f() $w_{1,2}^{(1)}$ $w_{2,1}^{(3)}$ $w_{2,2}^{(2)}$ $u_2^{(2)}$ $u_2^{(3)}$

Capa de entrada

Capa oculta

Capa de salida

Y



Luego la siguiente expresión para la derivada parcial con respecto a $w_{1,1}^{(1)}$ es seguir los dos caminos, el primer sumando es el primer camino rojo y se le suma el segundo camino rojo

$$\frac{\partial a_1^{(4)}}{\partial w_{1,1}^{(1)}} = a_1^{(4)} * \left(1 - a_1^{(4)}\right) * w_{1,1}^{(3)} * a_1^{(3)} * \left(1 - a_1^{(3)}\right) * w_{1,1}^{(2)} * a_1^{(2)} * \left(1 - a_1^{(2)}\right) * a_1^{(1)} + a_1^{(4)} * \left(1 - a_1^{(4)}\right) * w_{2,1}^{(3)} * a_2^{(3)} * \left(1 - a_2^{(3)}\right) * w_{1,2}^{(2)} * a_1^{(2)} * \left(1 - a_1^{(2)}\right) * a_1^{(1)}$$

Recomendado ir de la entrada a la salida para ver cómo se incrementa el nivel de las capas

$$\frac{\partial a_1^{(4)}}{\partial w_{1,1}^{(1)}} = a_1^{(1)} * a_1^{(2)} * \left(1 - a_1^{(2)}\right) * w_{1,1}^{(2)} * a_1^{(3)} * \left(1 - a_1^{(3)}\right) * w_{1,1}^{(3)} * a_1^{(4)} * \left(1 - a_1^{(4)}\right) + a_1^{(1)} * a_1^{(2)} * \left(1 - a_1^{(2)}\right) * w_{1,2}^{(2)} * a_2^{(3)} * \left(1 - a_2^{(3)}\right) * w_{2,1}^{(3)} * a_1^{(4)} * \left(1 - a_1^{(4)}\right)$$

Y así poder generalizar

$$\frac{\partial a_1^{(4)}}{\partial w_{1,1}^{(1)}} = a_1^{(1)} * a_1^{(2)} * \left(1 - a_1^{(2)}\right) * \left[\sum_{j=1}^{2} w_{1,j}^{(2)} * a_j^{(3)} * \left(1 - a_j^{(3)}\right) * w_{j,1}^{(3)}\right] * a_1^{(4)} * \left(1 - a_1^{(4)}\right)$$

La ventaja es que si las capas ocultas tienen más neuronas, sería cambiar el límite máximo en la sumatoria.

Renombrando la entrada y salida del perceptrón así:

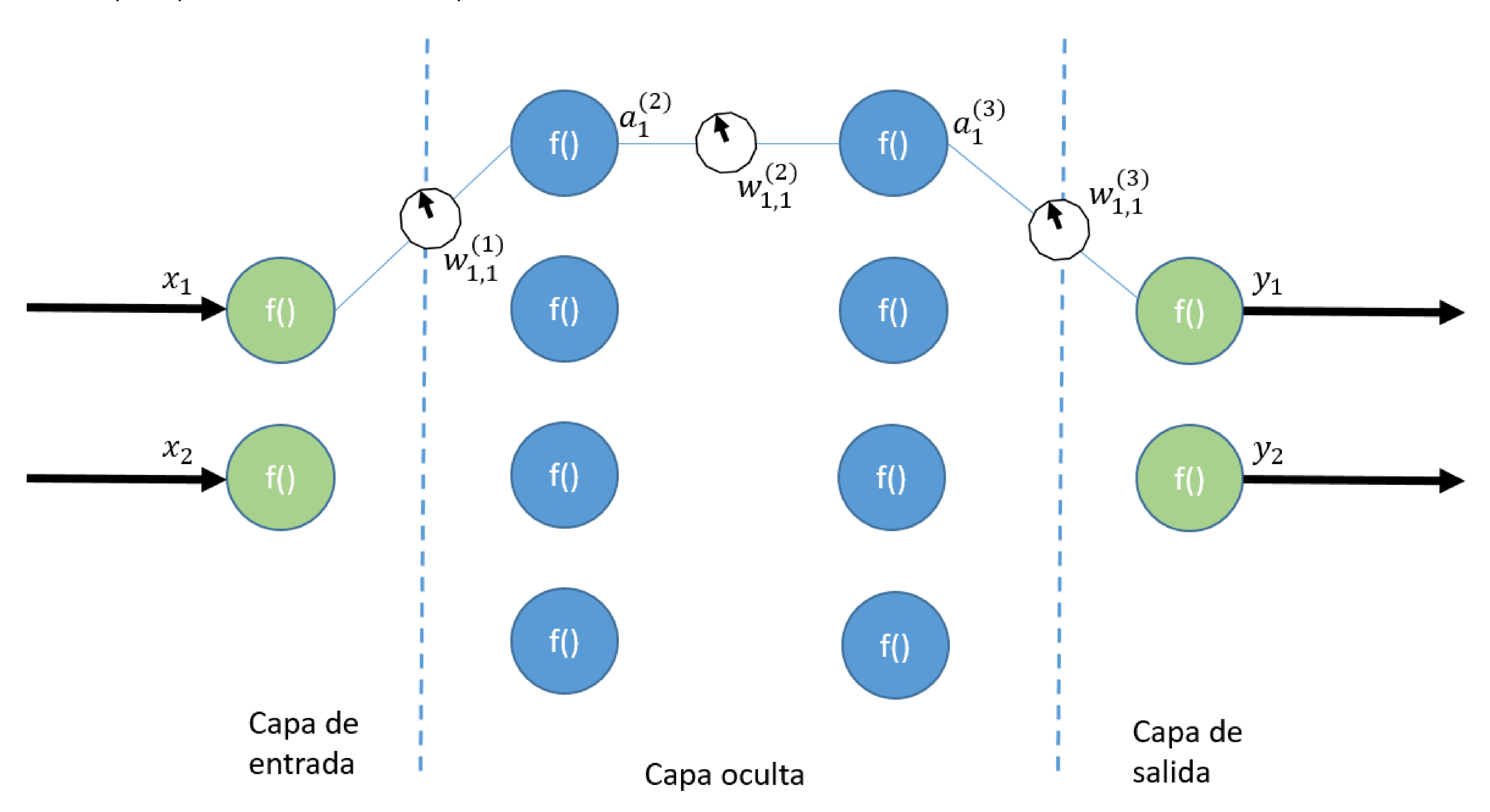
$$a_1^{(1)} = x_1$$

$$a_1^{(4)} = y_1$$

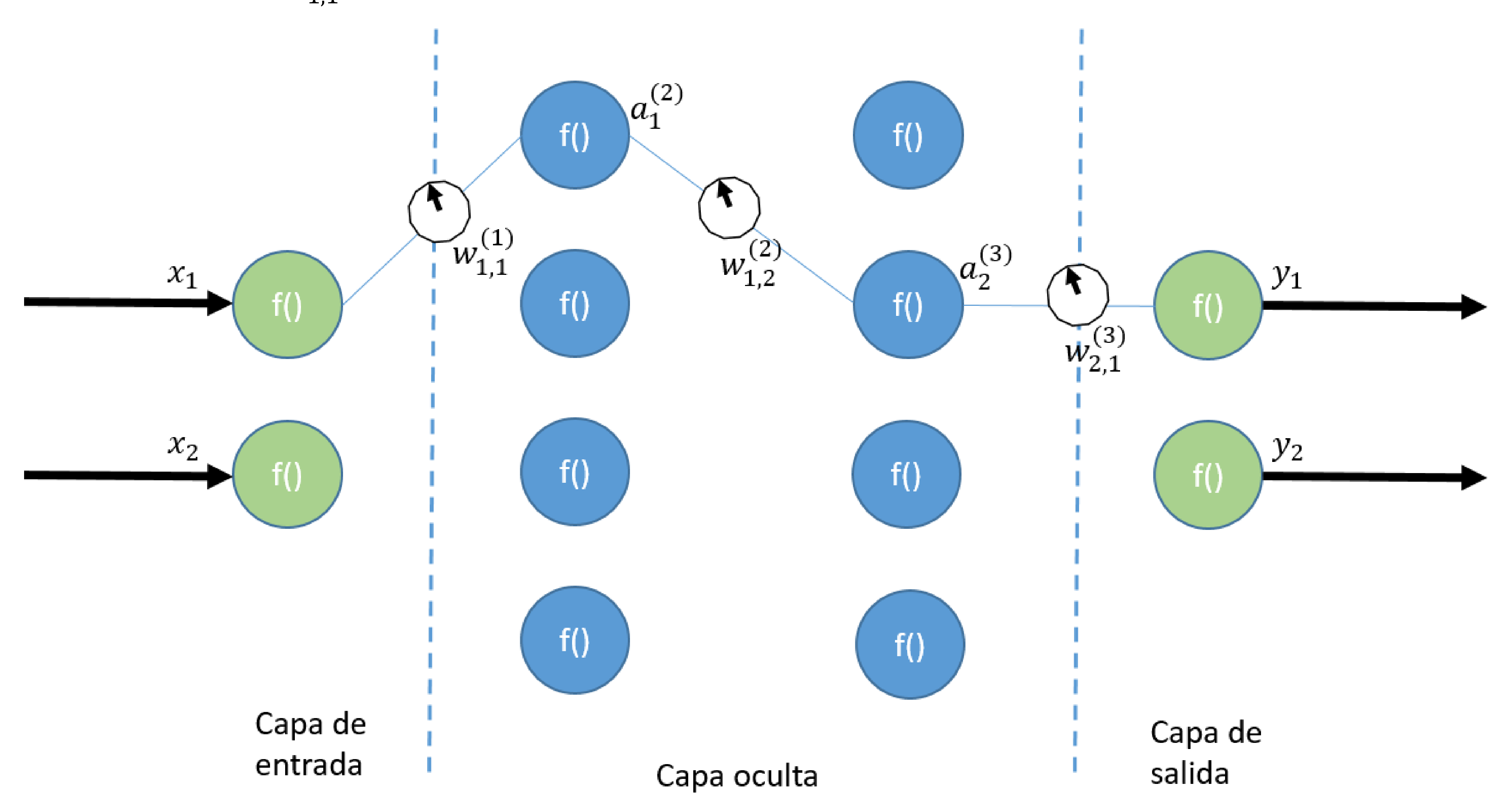
Entonces

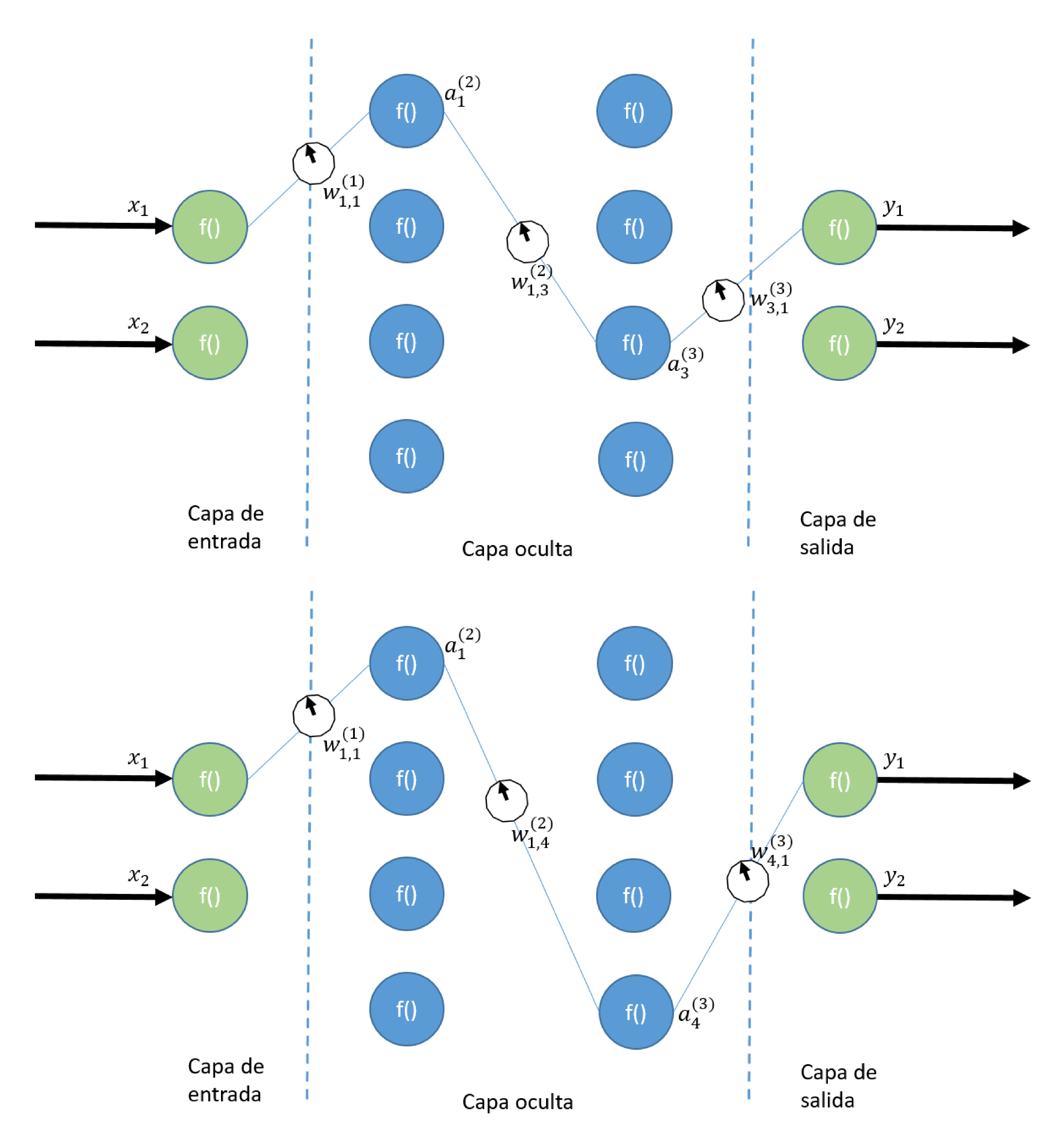
$$\frac{\partial y_1}{\partial w_{1,1}^{(1)}} = x_1 * a_1^{(2)} * \left(1 - a_1^{(2)}\right) * \left[\sum_{j=1}^{2} w_{1,j}^{(2)} * a_j^{(3)} * \left(1 - a_j^{(3)}\right) * w_{j,1}^{(3)}\right] * y_1 * (1 - y_1)$$

Con un perceptrón con más entradas y salidas como se ve a continuación



Si se desea dar con $\frac{\partial y_1}{\partial w_{1,1}^{(1)}}$, hay que considerar los diferentes caminos





Y de nuevo las capas que se nombran como n_1, n_2, n_3, n_4, donde: n_1 es la capa de entrada que no tiene procesamiento y tiene 2 neuronas. Ver:

n_1=2 (tiene dos neuronas)

n_2=4 (tiene cuatro neuronas)

n_3=4 (tiene cuatro neuronas)

n_4=2 (tiene dos neuronas)

Luego

$$\frac{\partial y_1}{\partial w_{1,1}^{(1)}} = x_1 * a_1^{(2)} * \left(1 - a_1^{(2)}\right) * \left[\sum_{p=1}^{n_3} w_{1,p}^{(2)} * a_p^{(3)} * \left(1 - a_p^{(3)}\right) * w_{p,1}^{(3)}\right] * y_1 * (1 - y_1)$$

Generalizando

$$\frac{\partial y_i}{\partial w_{j,k}^{(1)}} = x_j * a_k^{(2)} * \left(1 - a_k^{(2)}\right) * \left[\sum_{p=1}^{n_3} w_{k,p}^{(2)} * a_p^{(3)} * \left(1 - a_p^{(3)}\right) * w_{p,i}^{(3)}\right] * y_i * (1 - y_i)$$

Donde

i=1.. n_4

j=1.. n_1

k=1.. n_2

¿Y para los $w^{(2)}$?

$$\frac{\partial y_i}{\partial w_{j,k}^{(2)}} = a_j^{(2)} * a_k^{(3)} * \left(1 - a_k^{(3)}\right) * w_{k,i}^{(3)} * y_i * (1 - y_i)$$

¿Y para los $w^{(3)}$?

$$\frac{\partial y_i}{\partial w_{j,i}^{(3)}} = a_j^{(3)} * y_i * (1 - y_i)$$

¿Y los umbrales $u^{(2)}$?

$$\frac{\partial y_i}{\partial u_k^{(2)}} = 1 * a_k^{(2)} * \left(1 - a_k^{(2)}\right) * \left[\sum_{p=1}^{n_3} w_{k,p}^{(2)} * a_p^{(3)} * \left(1 - a_p^{(3)}\right) * w_{p,i}^{(3)}\right] * y_i * (1 - y_i)$$

Donde

i=1.. n_4

k=1.. n_2

¿Y los umbrales $u^{(3)}$?

$$\frac{\partial y_i}{\partial u_k^{(3)}} = 1 * a_k^{(3)} * \left(1 - a_k^{(3)}\right) * w_{k,i}^{(3)} * y_i * (1 - y_i)$$

¿Y los umbrales $u^{(4)}$?

$$\frac{\partial y_i}{\partial u_i^{(4)}} = 1 * y_i * (1 - y_i)$$

Tratamiento del error en el algoritmo de propagación hacia atrás

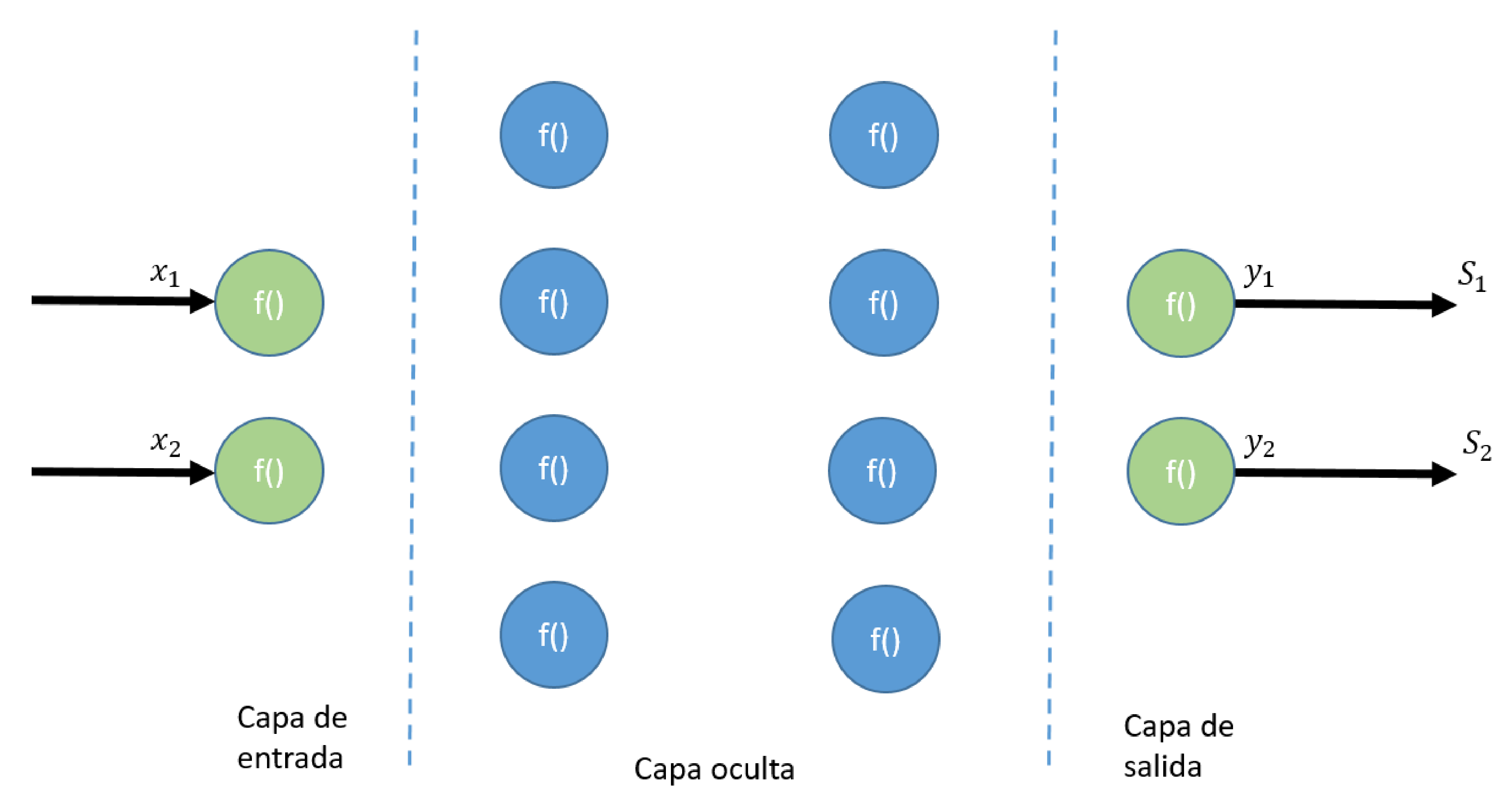
Tenemos la siguiente tabla

Entrada X_1	Entrada X_2	Valor esperado de salida S_1	Valor esperado de salida S_2
1	0	0	1
0	0	1	1
0	1	0	0

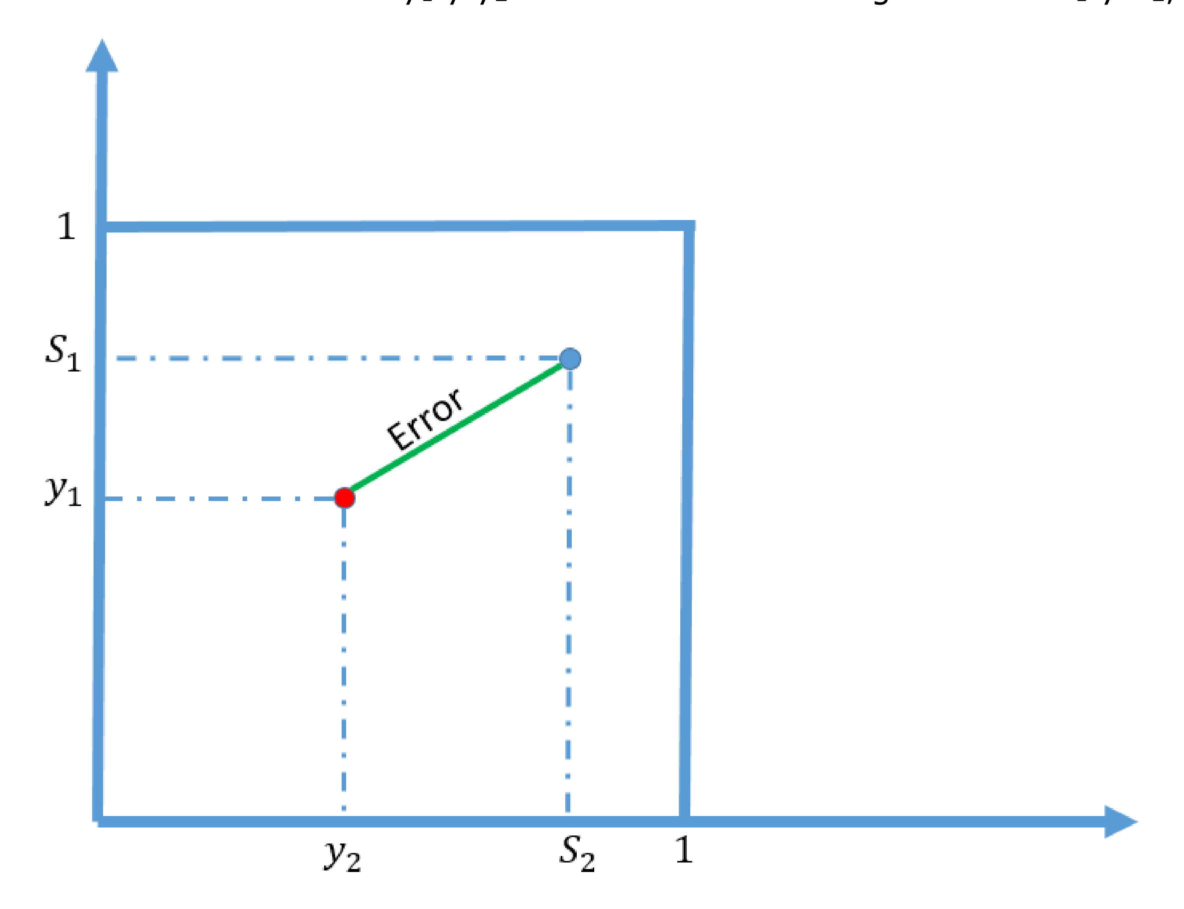
Pero en realidad estamos obteniendo con el perceptrón estas salidas

Entrada X_1	Entrada X_2	Salida real Y_1	Salida real Y_2
1	0	1	1
0	0	1	0
0	1	0	1

Hay un error evidente con las salidas porque no coinciden con lo esperado. ¿Qué hacer? Ajustar los pesos y los umbrales.



Si tomásemos las salidas y_1 y y_2 como coordenadas e igualmente S_1 y S_2, tendríamos lo siguiente:



Como la función de salida de las neuronas es la sigmoidea, la salida está entre 0 y 1.

En el gráfico de color verde está el error y para calcularlo es usar la fórmula de distancia entre dos puntos en un plano:

$$Error = \sqrt[2]{(S_2 - y_2)^2 + (S_1 - y_1)^2}$$

Requerimos minimizar ese Error, luego hay que considerar que dado:

$$f(x) = \sqrt[2]{g(x)}$$

Al derivar:

$$f'(x) = \frac{g'(x)}{2 * \sqrt[2]{g(x)}}$$

Y como hay que minimizar se iguala esa derivada a cero

$$f'(x) = \frac{g'(x)}{2 * \sqrt[2]{g(x)}} = 0$$

Luego

$$g'(x) = 0$$

En otras palabras, la raíz cuadrada de $f(x)$, es irrelevante cuando buscamos minimizar, porque lo importante es minimizar el interior. Luego la ecuación del error pasa a ser:

$$Error = (S_2 - y_2)^2 + (S_1 - y_1)^2$$

Que es más sencilla de evaluar. Pero aún no hemos terminado. El siguiente paso es multiplicarla por unas constantes quedando así:

$$Error = \frac{1}{2}(S_2 - y_2)^2 + \frac{1}{2}(S_1 - y_1)^2$$

¿Y por qué se hizo eso? Para hacer que la derivada de Error sea más sencilla. Y no hay que preocuparse porque afecte los resultados: como se busca minimizar, esas constantes no afectan el procedimiento.

¡OJO! Hay que recordar que y_1, y_2, varían, en cambio, S_1, S_2 son constantes porque son los valores esperados.

Hay que considerar esta regla matemática: Si P es una función con varias variables independientes, es decir: P(m,n) y Q también es otra función con esas mismas variables independientes, es decir: Q(m,n) y hay una *superfunción* que hace uso de P y Q, es decir: K(P,Q), entonces para derivar a K por una de las variables independientes, tenemos:

$$\frac{\partial K}{\partial m} = \frac{\partial K}{\partial P} * \frac{\partial P}{\partial m} + \frac{\partial K}{\partial Q} * \frac{\partial Q}{\partial m}$$

o

$$\frac{\partial K}{\partial n} = \frac{\partial K}{\partial P} * \frac{\partial P}{\partial n} + \frac{\partial K}{\partial Q} * \frac{\partial Q}{\partial n}$$

Luego

$$\frac{\partial Error}{\partial \blacksquare} = \frac{\partial Error}{\partial y_1} * \frac{\partial y_1}{\partial \blacksquare} + \frac{\partial Error}{\partial y_2} * \frac{\partial y_2}{\partial \blacksquare}$$

¿Y qué es ese cuadro relleno negro? Puede ser algún peso o algún umbral. Generalizando:

$$\frac{\partial Error}{\partial \blacksquare} = \sum_{i=1}^{n_4} \left(\frac{\partial Error}{\partial y_i} * \frac{\partial y_i}{\partial \blacksquare} \right)$$

Donde n_4 es el número de neuronas de la última capa.

Sabiendo que

$$Error = \frac{1}{2}\ (S_2 - y_2)^2 + \frac{1}{2}(S_1 - y_1)^2$$

Entonces la derivada de Error con respecto a y_1 es:

$$\frac{\partial Error}{\partial y_1} = y_1 - S_1$$

Generalizando

$$\frac{\partial Error}{\partial y_i} = y_i - S_i$$

Luego

$$\frac{\partial Error}{\partial \blacksquare} = \sum_{i=1}^{n_4} \left((y_i - S_i) * \frac{\partial y_i}{\partial \blacksquare} \right)$$

Queda entonces el cuadro relleno negro que como se mencionó anteriormente puede ser un peso o un umbral. Entonces si tenemos por ejemplo que:

$$\blacksquare = w_{j,i}^{(3)}$$

Entonces como hay una **i** en particular, la sumatoria se retira luego.

$$\frac{\partial Error}{\partial w_{j,i}^{(3)}} = (y_i - S_i) * \frac{\partial y_i}{\partial w_{j,i}^{(3)}}$$

Y como

$$\frac{\partial y_i}{\partial w_{j,i}^{(3)}} = a_j^{(3)} * y_i * (1 - y_i)$$

Luego

$$\frac{\partial Error}{\partial w_{j,i}^{(3)}} = (y_i - S_i) * a_j^{(3)} * y_i * (1 - y_i)$$

Ordenando

$$\frac{\partial Error}{\partial w_{j,i}^{(3)}} = a_j^{(3)} * (y_i - S_i) * y_i * (1 - y_i)$$

De nuevo la derivada del error

$$\frac{\partial Error}{\partial \blacksquare} = \sum_{i=1}^{n_4} \left((y_i - S_i) * \frac{\partial y_i}{\partial \blacksquare} \right)$$

Suponiendo que

$$\blacksquare = w_{j,k}^{(2)}$$

Entonces

$$\frac{\partial Error}{\partial w_{j,k}^{(2)}} = \sum_{i=1}^{n_4} \left((y_i - S_i) * \frac{\partial y_i}{\partial w_{j,k}^{(2)}} \right)$$

Y como

$$\frac{\partial y_i}{\partial w_{j,k}^{(2)}} = a_j^{(2)} * a_k^{(3)} * \left(1 - a_k^{(3)}\right) * w_{k,i}^{(3)} * y_i * (1 - y_i)$$

Entonces

$$\frac{\partial Error}{\partial w_{j,k}^{(2)}} = \sum_{i=1}^{n_4} \left((y_i - S_i) * a_j^{(2)} * a_k^{(3)} * \left(1 - a_k^{(3)}\right) * w_{k,i}^{(3)} * y_i * (1 - y_i) \right)$$

Simplificando

$$\frac{\partial Error}{\partial w_{j,k}^{(2)}} = a_j^{(2)} * a_k^{(3)} * \left(1 - a_k^{(3)}\right) * \sum_{i=1}^{n_4} \left((y_i - S_i) * w_{k,i}^{(3)} * y_i * (1 - y_i) \right)$$

De nuevo la derivada del error

$$\frac{\partial Error}{\partial \blacksquare} = \sum_{i=1}^{n_4} \left((y_i - S_i) * \frac{\partial y_i}{\partial \blacksquare} \right)$$

Suponiendo que

$$\blacksquare = w_{j,k}^{(1)}$$

Luego la derivada del error es:

$$\frac{\partial Error}{\partial w_{j,k}^{(1)}} = \sum_{i=1}^{n_4} \left((y_i - S_i) * \frac{\partial y_i}{\partial w_{j,k}^{(1)}} \right)$$

Y como se vio anteriormente que

$$\frac{\partial y_i}{\partial w_{j,k}^{(1)}} = x_j * a_k^{(2)} * \left(1 - a_k^{(2)}\right) * \left[\sum_{p=1}^{n_3} w_{k,p}^{(2)} * a_p^{(3)} * \left(1 - a_p^{(3)}\right) * w_{p,i}^{(3)} \right] * y_i * (1 - y_i)$$

Luego reemplazando en la expresión se obtiene:

$$\frac{\partial Error}{\partial w_{j,k}^{(1)}} = \sum_{i=1}^{n_4} \left((y_i - S_i) * x_j * a_k^{(2)} * \left(1 - a_k^{(2)}\right) * \left[\sum_{p=1}^{n_3} w_{k,p}^{(2)} * a_p^{(3)} * \left(1 - a_p^{(3)}\right) * w_{p,i}^{(3)} \right] * y_i * (1 - y_i) \right)$$

Simplificando

$$\frac{\partial Error}{\partial w_{j,k}^{(1)}} = x_j * a_k^{(2)} * \left(1 - a_k^{(2)}\right) * \sum_{i=1}^{n_4} \left((y_i - S_i) * \left[\sum_{p=1}^{n_3} w_{k,p}^{(2)} * a_p^{(3)} * \left(1 - a_p^{(3)}\right) * w_{p,i}^{(3)} \right] * y_i * (1 - y_i) \right)$$

$$\frac{\partial Error}{\partial w_{j,k}^{(1)}} = x_j * a_k^{(2)} * \left(1 - a_k^{(2)}\right) * \sum_{p=1}^{n_3} \left[w_{k,p}^{(2)} * a_p^{(3)} * \left(1 - a_p^{(3)}\right) * \sum_{i=1}^{n_4} \left(w_{p,i}^{(3)} * (y_i - S_i) * y_i * (1 - y_i) \right) \right]$$

En limpio las fórmulas para los pesos son:

$$\frac{\partial Error}{\partial w_{j,i}^{(3)}} = a_j^{(3)} * (y_i - S_i) * y_i * (1 - y_i)$$

$$\frac{\partial Error}{\partial w_{j,k}^{(2)}} = a_j^{(2)} * a_k^{(3)} * \left(1 - a_k^{(3)}\right) * \sum_{i=1}^{n_4} \left(w_{k,i}^{(3)} * (y_i - S_i) * y_i * (1 - y_i)\right)$$

$$\frac{\partial Error}{\partial w_{j,k}^{(1)}} = x_j * a_k^{(2)} * \left(1 - a_k^{(2)}\right) * \sum_{p=1}^{n_3} \left[w_{k,p}^{(2)} * a_p^{(3)} * \left(1 - a_p^{(3)}\right) * \sum_{i=1}^{n_4} \left(w_{p,i}^{(3)} * (y_i - S_i) * y_i * (1 - y_i)\right)\right]$$

Y para los umbrales sería:

$$\frac{\partial Error}{\partial u_i^{(4)}} = (y_i - S_i) * y_i * (1 - y_i)$$

$$\frac{\partial Error}{\partial u_k^{(3)}} = a_k^{(3)} * \left(1 - a_k^{(3)}\right) * \sum_{i=1}^{n_4} \left(w_{k,i}^{(3)} * (y_i - S_i) * y_i * (1 - y_i)\right)$$

$$\frac{\partial Error}{\partial u_k^{(2)}} = a_k^{(2)} * \left(1 - a_k^{(2)}\right) * \sum_{p=1}^{n_3} \left[w_{k,p}^{(2)} * a_p^{(3)} * \left(1 - a_p^{(3)}\right) * \sum_{i=1}^{n_4} \left(w_{p,i}^{(3)} * (y_i - S_i) * y_i * (1 - y_i)\right)\right]$$

Variando los pesos y umbrales con el algoritmo de propagación hacia atrás

La fórmula de variación de los pesos y umbrales es:

$$w_{j,i}^{(3)} \leftarrow w_{j,i}^{(3)} - \propto * \frac{\partial Error}{\partial w_{j,i}^{(3)}}$$

$$w_{j,k}^{(2)} \leftarrow w_{j,k}^{(2)} - \propto * \frac{\partial Error}{\partial w_{j,k}^{(2)}}$$

$$w_{j,k}^{(1)} \leftarrow w_{j,k}^{(1)} - \propto * \frac{\partial Error}{\partial w_{j,k}^{(1)}}$$

$$u_{i}^{(4)} \leftarrow u_{i}^{(4)} - \propto * \frac{\partial Error}{\partial u_{i}^{(4)}}$$

$$u_{k}^{(3)} \leftarrow u_{k}^{(3)} - \propto * \frac{\partial Error}{\partial u_{k}^{(3)}}$$

$$u_{k}^{(2)} \leftarrow u_{k}^{(2)} - \propto * \frac{\partial Error}{\partial u_{k}^{(2)}}$$

Donde $\propto$ es el factor de aprendizaje con un valor pequeño entre 0.1 y 0.9

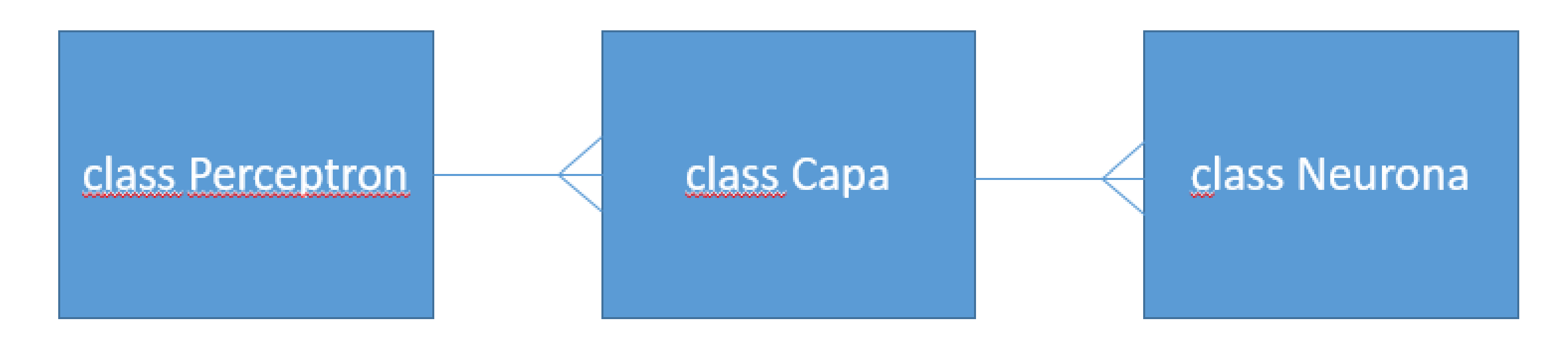
Implementación en C# del perceptrón multicapa

El siguiente modelo entidad-relación muestra cómo se compone un perceptrón multicapa



Un perceptrón tiene dos o más capas (mínimo una oculta y la de salida), no se considera la capa de entrada porque no hace operaciones. Una capa tiene uno o más neuronas.

Para implementarlo se hace uso de clases y listas.




```
using System;
using System.Collections.Generic;

class Perceptron {
    List<Capa> capas;
}

class Capa {
   List<Neurona> neuronas;
}

class Neurona {
}
```

Cada neurona tiene los pesos de entrada y el umbral. Para los pesos se hace uso de un arreglo unidimensional de tipo double

```
class Neurona {
    private double[] pesos; //Los pesos para cada entrada
    private double umbral;
}
```

En el constructor se inicializa el arreglo de pesos.

```
 class Neurona {
    private double[] pesos; //Los pesos para cada entrada
    private double umbral;

    public Neurona(Random azar, int TotalEntradas) { //Constructor
        pesos = new double[TotalEntradas];
        for (int cont=0; cont < TotalEntradas; cont++) pesos[cont] = azar.NextDouble();
        umbral = azar.NextDouble();
    }
}
```

El objeto azar es enviado al constructor de esta clase porque es necesario mantener sólo un generador de números pseudo-aleatorios.

Se añade a la clase neurona el método que calcula la salida que tiene como parámetro un arreglo unidimensional con los datos de entrada.

```
class Neurona {
    private double[] pesos; //Los pesos para cada entrada
    private double umbral;

    public Neurona(Random azar, int TotalEntradas) { //Constructor
        pesos = new double[TotalEntradas];
        for (int cont=0; cont < TotalEntradas; cont++) pesos[cont] = azar.NextDouble();
        umbral = azar.NextDouble();
    }

    public double calculaSalida(double[] entradas){
        double valor = 0;
        for (int cont = 0; cont < pesos.Length; cont++) valor += entradas[cont] * pesos[cont];
        valor += umbral;
        return 1 / (1 + Math.Exp(-valor));
    }
}
```

La clase capa crea las neuronas al instanciarse y guarda en un arreglo unidimensional, la salida de cada neurona de esa capa para facilitar los cálculos más adelante.

```
class Capa {
    List<Neurona> neuronas;
    public double[] salidas;

    public Capa(int totalNeuronas, int totalEntradas, Random azar){
        neuronas = new List<Neurona>();
        for (int genera = 1; genera <= totalNeuronas; genera++) neuronas.Add(new Neurona(azar, totalEntradas));
        salidas = new double[totalNeuronas];
    }
}
```

Se añade el método en que calcula la salida de cada neurona y guarda ese resultado en el arreglo unidimensional “salidas”

```
class Capa {
    List<Neurona> neuronas;
    public double[] salidas;

    public Capa(int totalNeuronas, int totalEntradas, Random azar){
        neuronas = new List<Neurona>();
        for (int genera = 1; genera <= totalNeuronas; genera++) neuronas.Add(new Neurona(azar, totalEntradas));
        salidas = new double[totalNeuronas];
    }

    public void CalculaCapa(double[] entradas){
        for (int cont = 0; cont < neuronas.Count; cont++)  salidas[cont] = neuronas[cont].calculaSalida(entradas);
    }
}
```

La clase Perceptron que debe crear las capas

```
class Perceptron {
    List<Capa> capas;

    //Crea las diversas capas
    public void creaCapas(int totalentradasexternas, int[] neuronasporcapa, Random azar){
        capas = new List<Capa>();
        capas.Add(new Capa(neuronasporcapa[0], totalentradasexternas, azar)); //La primera capa que hace cálculos
        for (int crea = 1; crea < neuronasporcapa.Length; crea++) capas.Add(new Capa(neuronasporcapa[crea], neuronasporcapa[crea-1], azar));
    }
}
```

El método “creaCapas” recibe tres parámetros:

totalentradasexternas: Es el número de entradas del exterior al perceptrón

neuronasporcapa: Es un arreglo unidimensional que tiene cuantas neuronas se crearán por cada capa

azar: El generador de números pseudo-aleatorios.

"calculaSalida" evalúa la capa inicial con las entradas externas al perceptrón, esa capa genera unos resultados que son almacenados en un arreglo unidimensional. Posteriormente se toma la segunda capa y se evalúa neurona por neurona tomando como entradas lo que produjo la capa inicial. Así sucesivamente.

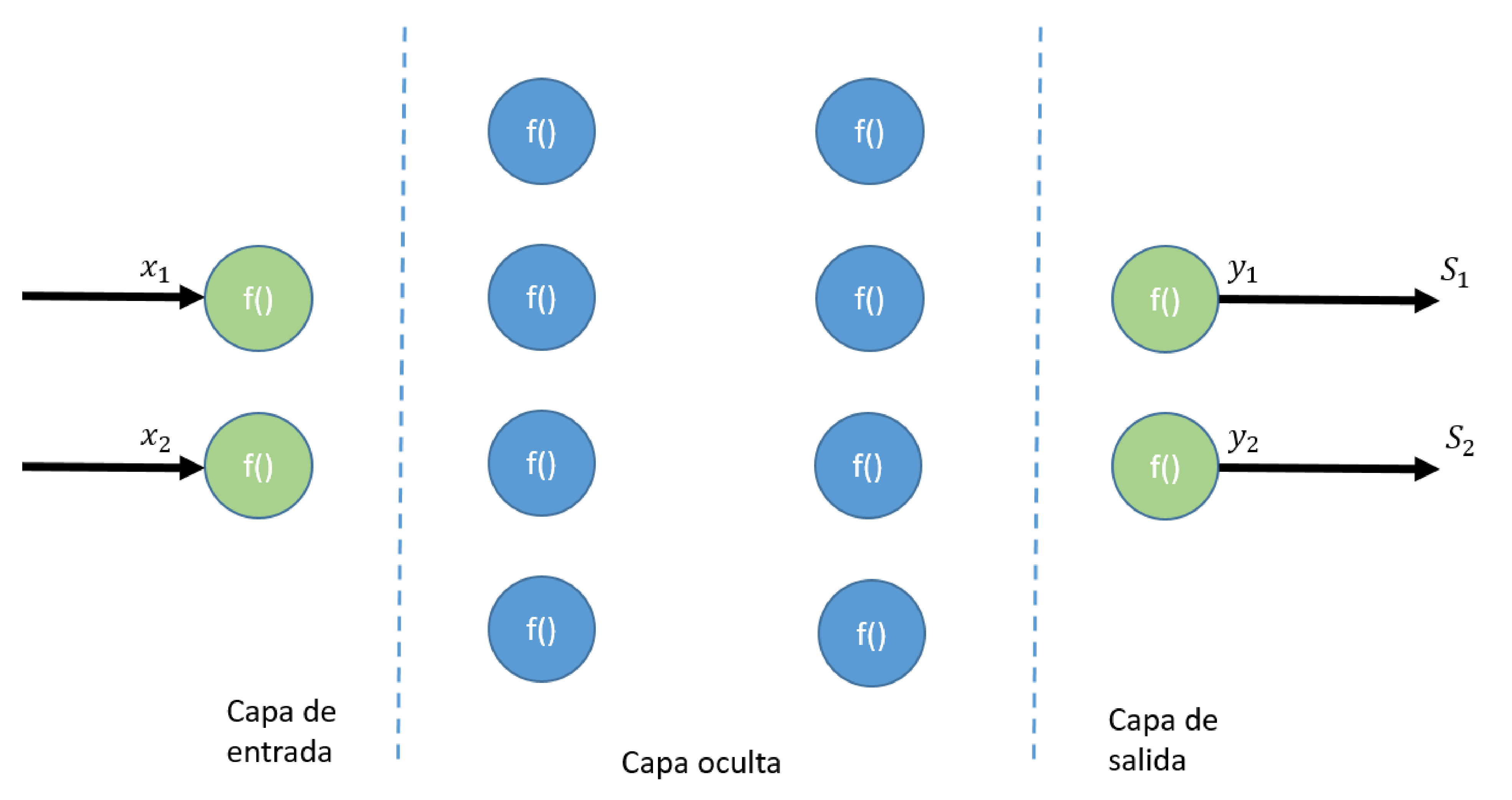
```
class Perceptron {
    List<Capa> capas;

    //Crea las diversas capas
    public void creaCapas(int totalentradasexternas, int[] neuronasporcapa, Random azar){
        capas = new List<Capa>();
        capas.Add(new Capa(neuronasporcapa[0], totalentradasexternas, azar)); //La primera capa que hace cálculos
        for (int crea = 1; crea < neuronasporcapa.Length; crea++) capas.Add(new Capa(neuronasporcapa[crea], neuronasporcapa[crea-1], azar));
    }

    //Hace el cálculo
    public void calculaSalida(double[] entradas)
    {
        //La primera capa recibe las entradas externas
        capas[0].CalculaCapa(entradas);

        //Las siguientes capas, hacen los cálculos usando como entrada la salida de la capa anterior
        for (int cont = 1; cont < capas.Count; cont++) {
            capas[cont].CalculaCapa(capas[cont - 1].salidas);
        }
    }
}
```

Ejemplo de uso de la clase Perceptron para generar este diseño en particular:




```
class Program {
    static void Main(string[] args)
    {
        Random azar = new Random(); //Un solo generador de números pseudo-aleatorios
        Perceptron perceptron = new Perceptron();

        //Número de neuronas que tendrá cada capa
        int[] neuronasporcapa = new int[3];
        neuronasporcapa[0] = 4; //La capa inicial tendrá 4 neuronas
        neuronasporcapa[1] = 4; //La segunda capa tendrá 4 neuronas
        neuronasporcapa[2] = 2; //La capa final tendrá 2 neuronas

        //Habrán 2 entradas externas, se envía el arreglo de neuronas por capa y el generador de números pseudo-aleatorios
        perceptron.creaCapas(2, neuronasporcapa, azar);

        //Estas serán las entradas externas al perceptrón
        double[] entradas = new double[2];
        entradas[0] = 1;
        entradas[1] = 0;

        //Se hace el cálculo
        perceptron.calculaSalida(entradas);

        Console.ReadKey();
    }
}
```

Es en el programa principal que se crea el objeto que genera los números pseudo-aleatorios y se le envía al perceptrón.

Para mostrar cómo opera el perceptrón se muestra una hoja en Excel

Entradas			Capa 2	Peso 1	Peso 2			Umbral	Valor	Salida
X1	1		Neurona 1	0,47247986	0,678516118			0,80741687	1,2799	0,782
X2	0		Neurona 2	0,801949133	0,779935713			0,204385216	1,00633	0,732
			Neurona 3	0,184836394	0,39429444			0,679925648	0,86476	0,704
			Neurona 4	0,888175346	0,885816893			0,744964491	1,63314	0,837
			Capa 3	**Peso 1**	**Peso 2**	**Peso 3**	**Peso 4**	**Umbral**	**Valor**	**Salida**
			Neurona 1	0,857642265	0,48017831	0,318394323	0,170780095	0,192821788	1,582	0,830
			Neurona 2	0,339199577	0,397973949	0,102941101	0,399645193	0,039451026	1,003	0,732
			Neurona 3	0,815089183	0,745557561	0,140308591	0,816845513	0,076225015	2,042	0,885
			Neurona 4	0,751872803	0,543341596	0,431814902	0,189449635	0,496551805	1,945	0,875
			Capa 4	**Peso 1**	**Peso 2**	**Peso 3**	**Peso 4**	**Umbral**	**Valor**	**Salida**
			Neurona 1	0,034647338	0,272564194	0,046934236	0,596919764	0,234838131	1,027	0,736
			Neurona 2	0,897890884	0,010213592	0,021097544	0,229079776	0,094218275	1,066	0,744

Por dentro está así realizada la hoja electrónica: Los pesos y los umbrales son aleatorios =ALEATORIO()

SUMA | =ALEATORIO()

	A	B	C	D	E	F	G	H	I	J	K
1	Entradas			**Capa 2**	**Peso 1**	**Peso 2**			**Umbral**	**Valor**	**Salida**
2	X1	1		Neurona 1	0,47247986	=ALEATORIO()			0,80741687	1,2799	0,782
3	X2	0		Neurona 2	0,801949133	0,779935713			0,204385216	1,00633	0,732
4				Neurona 3	0,184836394	0,39429444			0,679925648	0,86476	0,704
5				Neurona 4	0,888175346	0,885816893			0,744964491	1,63314	0,837
6											
7				**Capa 3**	**Peso 1**	**Peso 2**	**Peso 3**	**Peso 4**	**Umbral**	**Valor**	**Salida**
8				Neurona 1	0,857642265	0,48017831	0,318394323	0,170780095	0,192821788	1,582	0,830
9				Neurona 2	0,339199577	0,397973949	0,102941101	0,399645193	0,039451026	1,003	0,732
10				Neurona 3	0,815089183	0,745557561	0,140308591	0,816845513	0,076225015	2,042	0,885
11				Neurona 4	0,751872803	0,543341596	0,431814902	0,189449635	0,496551805	1,945	0,875
12											
13				**Capa 4**	**Peso 1**	**Peso 2**	**Peso 3**	**Peso 4**	**Umbral**	**Valor**	**Salida**
14				Neurona 1	0,034647338	0,272564194	0,046934236	0,596919764	0,234838131	1,027	0,736
15				Neurona 2	0,897890884	0,010213592	0,021097544	0,229079776	0,094218275	1,066	0,744

La columna J con el nombre de valor es el cálculo de las entradas por los pesos adicionándole el umbral

	A	B	C	D	E	F	G	H	I	J	K	L
1	Entradas			**Capa 2**	**Peso 1**	**Peso 2**			**Umbral**	**Valor**	**Salida**	
2	X1	1		Neurona 1	0,47247986	0,678516118			0,80741687	1,2799	0,782	
3	X2	0		Neurona 2	0,801949133	0,779935713			0,204385216	=E3*$B$2+F3*$B$3+I3		
4				Neurona 3	0,184836394	0,39429444			0,679925648	0,86476	0,704	
5				Neurona 4	0,888175346	0,885816893			0,744964491	1,63314	0,837	
6												
7				**Capa 3**	**Peso 1**	**Peso 2**	**Peso 3**	**Peso 4**	**Umbral**	**Valor**	**Salida**	
8				Neurona 1	0,857642265	0,48017831	0,318394323	0,170780095	0,192821788	1,582	0,830	
9				Neurona 2	0,339199577	0,397973949	0,102941101	0,399645193	0,039451026	1,003	0,732	
10				Neurona 3	0,815089183	0,745557561	0,140308591	0,816845513	0,076225015	2,042	0,885	
11				Neurona 4	0,751872803	0,543341596	0,431814902	0,189449635	0,496551805	1,945	0,875	
12												
13				**Capa 4**	**Peso 1**	**Peso 2**	**Peso 3**	**Peso 4**	**Umbral**	**Valor**	**Salida**	
14				Neurona 1	0,034647338	0,272564194	0,046934236	0,596919764	0,234838131	1,027	0,736	
15				Neurona 2	0,897890884	0,010213592	0,021097544	0,229079776	0,094218275	1,066	0,744	
16												

La columna K con el nombre de la salida es el cálculo de la función sigmoidea de la neurona

	A	B	C	D	E	F	G	H	I	J	K	L
1	Entradas			**Capa 2**	**Peso 1**	**Peso 2**			**Umbral**	**Valor**	**Salida**	
2	X1	1		Neurona 1	0,47247986	0,678516118			0,80741687	1,2799	0,782	
3	X2	0		Neurona 2	0,801949133	0,779935713			0,204385216	1,00633	0,732	
4				Neurona 3	0,184836394	0,39429444			0,679925648	0,86476	=1/(1+EXP(-J4))	
5				Neurona 4	0,888175346	0,885816893			0,744964491	1,63314	0,837	
6												
7				**Capa 3**	**Peso 1**	**Peso 2**	**Peso 3**	**Peso 4**	**Umbral**	**Valor**	**Salida**	
8				Neurona 1	0,857642265	0,48017831	0,318394323	0,170780095	0,192821788	1,582	0,830	
9				Neurona 2	0,339199577	0,397973949	0,102941101	0,399645193	0,039451026	1,003	0,732	
10				Neurona 3	0,815089183	0,745557561	0,140308591	0,816845513	0,076225015	2,042	0,885	
11				Neurona 4	0,751872803	0,543341596	0,431814902	0,189449635	0,496551805	1,945	0,875	
12												
13				**Capa 4**	**Peso 1**	**Peso 2**	**Peso 3**	**Peso 4**	**Umbral**	**Valor**	**Salida**	
14				Neurona 1	0,034647338	0,272564194	0,046934236	0,596919764	0,234838131	1,027	0,736	
15				Neurona 2	0,897890884	0,010213592	0,021097544	0,229079776	0,094218275	1,066	0,744	
16												

Observamos que en la Capa 3 que el valor (columna J) se calcula con la salida de la Capa 2

	A	B	C	D	E	F	G	H	I	J	K
1	Entradas			**Capa 2**	**Peso 1**	**Peso 2**			**Umbral**	**Valor**	**Salida**
2	X1	1		Neurona 1	0,47247986	0,678516118			0,80741687	1,2799	0,782
3	X2	0		Neurona 2	0,801949133	0,779935713			0,204385216	1,00633	0,732
4				Neurona 3	0,184836394	0,39429444			0,679925648	0,86476	0,704
5				Neurona 4	0,888175346	0,885816893			0,744964491	1,63314	0,837
6											
7				**Capa 3**	**Peso 1**	**Peso 2**	**Peso 3**	**Peso 4**	**Umbral**	**Valor**	**Salida**
8				Neurona 1	0,857642265	0,48017831	0,318394323	0,170780095	0,192821788	1,582	0,830
9				Neurona 2	0,339199577	0,397973949	0,102941101	0,399645193	0,039451026	=$K$2*E9+$K$3*F9+$K$4*G9+$K$5*H9+I9	
10				Neurona 3	0,815089183	0,745557561	0,140308591	0,816845513	0,076225015	2,042	0,885
11				Neurona 4	0,751872803	0,543341596	0,431814902	0,189449635	0,496551805	1,945	0,875
12											
13				**Capa 4**	**Peso 1**	**Peso 2**	**Peso 3**	**Peso 4**	**Umbral**	**Valor**	**Salida**
14				Neurona 1	0,034647338	0,272564194	0,046934236	0,596919764	0,234838131	1,027	0,736
15				Neurona 2	0,897890884	0,010213592	0,021097544	0,229079776	0,094218275	1,066	0,744
16											
17											

Observamos que en la Capa 4 que el valor (columna J) se calcula con la salida de la Capa 3

	A	B	C	D	E	F	G	H	I	J	K
1	Entradas			**Capa 2**	**Peso 1**	**Peso 2**			**Umbral**	**Valor**	**Salida**
2	X1	1		Neurona 1	0,47247986	0,678516118			0,80741687	1,2799	0,782
3	X2	0		Neurona 2	0,801949133	0,779935713			0,204385216	1,00633	0,732
4				Neurona 3	0,184836394	0,39429444			0,679925648	0,86476	0,704
5				Neurona 4	0,888175346	0,885816893			0,744964491	1,63314	0,837
6											
7				**Capa 3**	**Peso 1**	**Peso 2**	**Peso 3**	**Peso 4**	**Umbral**	**Valor**	**Salida**
8				Neurona 1	0,857642265	0,48017831	0,318394323	0,170780095	0,192821788	1,582	0,830
9				Neurona 2	0,339199577	0,397973949	0,102941101	0,399645193	0,039451026	1,003	0,732
10				Neurona 3	0,815089183	0,745557561	0,140308591	0,816845513	0,076225015	2,042	0,885
11				Neurona 4	0,751872803	0,543341596	0,431814902	0,189449635	0,496551805	1,945	0,875
12											
13				**Capa 4**	**Peso 1**	**Peso 2**	**Peso 3**	**Peso 4**	**Umbral**	**Valor**	**Salida**
14				Neurona 1	0,034647338	0,272564194	0,046934236	0,596919764	0,234838131	=$K$8*E14+$K$9*F14+$K$10*G14+$K$11*H14+I14	
15				Neurona 2	0,897890884	0,010213592	0,021097544	0,229079776	0,094218275	1,066	0,744
16											

Se modifica el código en C# para que muestre paso a paso como hace los cálculos por capa y por neurona. Este sería el código completo:

```
using System;
using System.Collections.Generic;

namespace PerceptronMultiCapa {
    class Neurona {
        private double[] pesos; //Los pesos para cada entrada
        private double umbral;

        public Neurona(Random azar, int TotalEntradas) { //Constructor
            pesos = new double[TotalEntradas];
            for (int cont=0; cont < TotalEntradas; cont++) pesos[cont] = azar.NextDouble();
            umbral = azar.NextDouble();
        }

        public double calculaSalida(double[] entradas){ //Calcula la salida de la neurona
            ImprimePesosUmbral();
            double valor = 0;
            for (int cont = 0; cont < pesos.Length; cont++) valor += entradas[cont] * pesos[cont];
            valor += umbral;
            return 1 / (1 + Math.Exp(-valor)); //Función sigmoidea
        }

        public void ImprimePesosUmbral(){
            Console.Write("\nPesos: ");
            for (int cont = 0; cont < pesos.Length; cont++) Console.Write("{0:F4}; ", pesos[cont]);
            Console.Write(" Umbral: {0:F4}", umbral);
        }
    }

    class Capa {
        List<Neurona> neuronas;
        public double[] salidas;

        public Capa(int totalNeuronas, int totalEntradas, Random azar){ //Constructor
            neuronas = new List<Neurona>();
            for (int genera = 1; genera <= totalNeuronas; genera++) neuronas.Add(new Neurona(azar, totalEntradas));
            salidas = new double[totalNeuronas];
        }

        public void CalculaCapa(double[] entradas){ //Calcula las salidas de la capa
            Console.Write("Entra: ");
            for (int cont2 = 0; cont2 < entradas.Length; cont2++) Console.Write("{0:F4}; ", entradas[cont2]);
            for (int cont = 0; cont < neuronas.Count; cont++)  salidas[cont] = neuronas[cont].calculaSalida(entradas);
            Console.WriteLine(" ");
            Console.Write("Salir: ");
            for (int cont2 = 0; cont2 < salidas.Length; cont2++) Console.Write("{0:F4}; ", salidas[cont2]);
            Console.WriteLine(" ");
        }
    }

    class Perceptron {
        List<Capa> capas;

        //Crea las diversas capas
        public void creaCapas(int totalentradasexternas, int[] neuronasporcapa, Random azar){
            capas = new List<Capa>();
            capas.Add(new Capa(neuronasporcapa[0], totalentradasexternas, azar)); //La primera capa que hace cálculos
            for (int crea = 1; crea < neuronasporcapa.Length; crea++) capas.Add(new Capa(neuronasporcapa[crea], neuronasporcapa[crea-1], azar));
        }

        //Hace el cálculo
        public void calculaSalida(double[] entradas)
        {
            //La primera capa recibe las entradas externas
            Console.WriteLine("============================ Capa 2 ============================ ");
            capas[0].CalculaCapa(entradas);

            //Las siguientes capas, hacen los cálculos usando como entrada la salida de la capa anterior
            for (int cont = 1; cont < capas.Count; cont++) {
                Console.WriteLine("============================  Capa " + (cont+2).ToString() + " ============================ ");
                capas[cont].CalculaCapa(capas[cont - 1].salidas);
            }
        }
    }

    class Program {
        static void Main(string[] args)
        {
            Random azar = new Random(); //Un solo generador de números pseudo-aleatorios
            Perceptron perceptron = new Perceptron();

            //Número de neuronas que tendrá cada capa
            int[] neuronasporcapa = new int[3];
            neuronasporcapa[0] = 4; //La capa inicial tendrá 4 neuronas
            neuronasporcapa[1] = 4; //La segunda capa tendrá 4 neuronas
            neuronasporcapa[2] = 2; //La capa final tendrá 2 neuronas

            //Habrán 2 entradas externas, se envía el arreglo de neuronas por capa y el generador de números pseudo-aleatorios
            perceptron.creaCapas(2, neuronasporcapa, azar);

            //Estas serán las entradas externas al perceptrón
            double[] entradas = new double[2];
            entradas[0] = 1;
            entradas[1] = 0;
```

```
            //Se hace el cálculo
            perceptron.calculaSalida(entradas);

            Console.ReadKey();
        }
    }
}
```

Ejemplo de ejecución

```
file:///C:/Users/engin/onedrive/documentos/visual studio 2015/Projects/RedNeuronal/RedNeuronal/bin/Debug/RedNeuronal.EXE
============================ Capa 2 ============================
Entra: 1,0000; 0,0000;
Pesos: 0,9123; 0,3673;  Umbral: 0,8274
Pesos: 0,6570; 0,7744;  Umbral: 0,7652
Pesos: 0,7321; 0,0866;  Umbral: 0,3804
Pesos: 0,0836; 0,1081;  Umbral: 0,2376
Salir: 0,8507; 0,8057; 0,7526; 0,5796;
============================  Capa 3 ============================
Entra: 0,8507; 0,8057; 0,7526; 0,5796;
Pesos: 0,8175; 0,0565; 0,6797; 0,1879;  Umbral: 0,1368
Pesos: 0,3356; 0,8859; 0,1251; 0,0257;  Umbral: 0,7347
Pesos: 0,2560; 0,7488; 0,3905; 0,1533;  Umbral: 0,3677
Pesos: 0,9230; 0,7901; 0,1115; 0,1521;  Umbral: 0,7371
Salir: 0,8173; 0,8633; 0,8280; 0,9114;
============================  Capa 4 ============================
Entra: 0,8173; 0,8633; 0,8280; 0,9114;
Pesos: 0,3810; 0,9489; 0,8371; 0,7926;  Umbral: 0,4356
Pesos: 0,7818; 0,1284; 0,5702; 0,1797;  Umbral: 0,6798
Salir: 0,9517; 0,8875;
```

Comparando con la hoja de cálculo (se copian los valores de pesos y umbrales)

Entradas			**Capa 2**	**Peso 1**	**Peso 2**			**Umbral**	**Valor**	**Salida**
X1	1		Neurona 1	0,9123	0,3673			0,8274	1,7397	0,8506
X2	0		Neurona 2	0,657	0,7744			0,7652	1,4222	0,8057
			Neurona 3	0,7321	0,0866			0,3804	1,1125	0,7526
			Neurona 4	0,0836	0,1081			0,2376	0,3212	0,5796
			Capa 3	**Peso 1**	**Peso 2**	**Peso 3**	**Peso 4**	**Umbral**	**Valor**	**Salida**
			Neurona 1	0,8175	0,0565	0,6797	0,1879	0,1368	1,498	0,8173
			Neurona 2	0,3356	0,8859	0,1251	0,0257	0,7347	1,843	0,8633
			Neurona 3	0,256	0,7488	0,3905	0,1533	0,3677	1,572	0,8280
			Neurona 4	0,923	0,7901	0,1115	0,1521	0,7371	2,331	0,9114
			Capa 4	**Peso 1**	**Peso 2**	**Peso 3**	**Peso 4**	**Umbral**	**Valor**	**Salida**
			Neurona 1	0,381	0,9489	0,8371	0,7926	0,4356	2,982	0,9517
			Neurona 2	0,7818	0,1284	0,5702	0,1797	0,6798	2,066	0,8875

Nota: en el programa en C# los valores están formateados para mostrar 4 decimales, pero internamente el cálculo se hace con todos los decimales. Por eso hay una ligera variación en la primera salida de la capa 1 de Excel con el de C#. Salvo eso, las salidas coinciden.

Mejorando la implementación en C#

Mientras implementaba el algoritmo me percaté de un problema que inicia siendo molesto y termina confundiendo. En las páginas anteriores se hace la deducción matemática de las fórmulas y se considera que las capas se enumeran desde 1 hasta 4. Allí tenemos un inconveniente con C#, porque al crear las capas, estas inician con el índice cero. Y no solo eso, la Capa 1 es una capa que no hace ningún tratamiento matemático, es una capa comodín para mencionar las entradas externas al perceptrón. Por esta razón tampoco se crea esta capa en C#. Entonces al mencionar la capa 4, en el código en C# sería capa[2].

Problema similar con las neuronas, que al ir en listas, inician con el índice cero. Luego nombrar los pesos y umbrales se torna en una labor de traducción a C# no intuitiva. ¿Cómo referirse a $w_{1,3}^{(2)}$? En la implementación en C# es perceptrón.capa[0].neurona[0].peso[2]

Para hacer más intuitivo el código en C# y coincida con lo expuesto en las fórmulas matemáticas, los índices iniciarán en 1, las capas se numerarán desde la 1 (así no tenga procesamiento esta capa en particular) y se hará uso de matrices y vectores. Este es el nuevo código:

```
using System;

namespace RedNeuronal2 {
    class Program {
        static void Main(string[] args)
        {
            //Los pesos serán arreglos multidimensionales (3 dimensiones). Así: W[capa, neurona inicial, neurona final]
            double[,,] W;

            //Las salidas de cada neurona serán arreglos bidimensionales. Así: A[capa, neurona que produce la salida]
            double[,] A;

            //Los umbrales de cada neurona serán arreglos bidimensionales. Así: U[capa, neurona que produce la salida]
            double[,] U;

            //Las entradas al perceptrón
            int TotalEntradasExternas = 2;
            double[] E = new double[TotalEntradasExternas+1];
            E[1] = 1;
            E[2] = 0;

            int TotalCapas = 4; //Total capas que tendrá el perceptrón
            int[] neuronasporcapa = new int[TotalCapas+1]; //Los índices iniciarán en 1 en esta implementación
            neuronasporcapa[1] = TotalEntradasExternas; //Entradas externas del perceptrón
            neuronasporcapa[2] = 4; //Capa oculta con 4 neuronas
            neuronasporcapa[3] = 4; //Capa oculta con 4 neuronas
            neuronasporcapa[4] = 2; //Capa de salida con 2 neuronas

            //Detecta el máximo número de neuronas por capa para dimensionar los arreglos
            int maxNeuronas = 0;
            for (int capa = 1; capa <= TotalCapas; capa++) if (neuronasporcapa[capa] > maxNeuronas) maxNeuronas = neuronasporcapa[capa];
            W = new double[TotalCapas + 1, maxNeuronas + 1, maxNeuronas + 1];
            A = new double[TotalCapas + 1, maxNeuronas + 1];
            U = new double[TotalCapas + 1, maxNeuronas + 1];

            //Entradas del perceptron pasan a la salida de la primera capa
            for (int cont = 1; cont <= TotalEntradasExternas; cont++) A[1, cont] = E[cont];

            //Da valores aleatorios a pesos y umbrales
            Random azar = new Random(10);

            for (int capa = 1; capa <= TotalCapas; capa++)
                for (int i = 1; i <= maxNeuronas; i++)
                    for (int j = 1; j <= maxNeuronas; j++){
                        W[capa, i, j] = azar.NextDouble();
                        Console.WriteLine("W[" + capa.ToString() + "," + i.ToString() + "," + j.ToString() + "]=" + W[capa, i, j].ToString());
                    }

            for (int capa = 1; capa <= TotalCapas; capa++)
                for (int neurona = 1; neurona <= maxNeuronas; neurona++){
                    U[capa, neurona] = azar.NextDouble();
                    Console.WriteLine("U[" + capa.ToString() + "," + neurona.ToString() + "]=" + U[capa, neurona].ToString());
                }

            //Procesando
            Console.WriteLine("Procesando:");
            for (int capa = 2; capa <= TotalCapas; capa++)
                for (int cont = 1; cont <= neuronasporcapa[capa]; cont++) {
                    A[capa, cont] = 0;
                    for (int entra = 1; entra <= neuronasporcapa[capa-1]; entra++) //La capa anterior
                        A[capa, cont] += A[capa-1, entra] * W[capa-1, entra, cont];
                    A[capa, cont] += U[capa, cont];
                    A[capa, cont] = 1 / (1 + Math.Exp(-A[capa, cont]));
                }

            Console.WriteLine(A[2, 1].ToString() + " ; " + A[2, 2].ToString() + " ; " + A[2, 3].ToString() + " ; " + A[2, 4].ToString() + " ; ");
            Console.WriteLine(A[3, 1].ToString() + " ; " + A[3, 2].ToString() + " ; " + A[3, 3].ToString() + " ; " + A[3, 4].ToString() + " ; ");
            Console.WriteLine(A[4, 1].ToString() + " ; " + A[4, 2].ToString());

            Console.ReadKey();
        }
    }
}
```

Algoritmo de retro propagación en C#

Las fórmulas del algoritmo de retro propagación son:

$$\frac{\partial Error}{\partial w_{j,i}^{(3)}} = a_j^{(3)} * (y_i - S_i) * y_i * (1 - y_i)$$

Y

$$w_{j,i}^{(3)} \leftarrow w_{j,i}^{(3)} - \propto * \frac{\partial Error}{\partial w_{j,i}^{(3)}}$$

Se implementa así:

```
            //Salidas esperadas
            double[] S = new double[3];
            S[1] = 0;
            S[2] = 1;

            //Factor de aprendizaje
            double alpha = 0.4;

            //Procesa capa 4
            for (int j=1; j <= neuronasporcapa[3]; j++)
                for (int i=1; i <= neuronasporcapa[4]; i++)
                {
                    double Yi = A[4, i];
                    double dE3 = A[3, j] * (Yi - S[i]) * Yi * (1 - Yi);
                    Console.WriteLine("dError/dW(3)" + j.ToString() + "," + i.ToString() + " = " + dE3.ToString());
                    double nuevoPeso = W[3, j, i] - alpha * dE3;
                    Console.WriteLine("W(3)" + j.ToString() + "," + i.ToString() + " = " + nuevoPeso.ToString());
                }
```

La capa anterior

$$\frac{\partial Error}{\partial w_{j,k}^{(2)}} = a_j^{(2)} * a_k^{(3)} * \left(1 - a_k^{(3)}\right) * \sum_{i=1}^{n_4} \left(w_{k,i}^{(3)} * (y_i - S_i) * y_i * (1 - y_i)\right)$$

Y

$$w_{j,k}^{(2)} \leftarrow w_{j,k}^{(2)} - \propto * \frac{\partial Error}{\partial w_{j,k}^{(2)}}$$

Se implementa así:

```
            //Procesa capa 3
            for (int j = 1; j <= neuronasporcapa[2]; j++)
                for (int k = 1; k <= neuronasporcapa[3]; k++)
                {
                    double acum = 0;
                    for (int i = 1; i <= neuronasporcapa[4]; i++)
                    {
                        double Yi = A[4, i];
                        acum += W[3, k, i] * (Yi - S[i]) * Yi * (1 - Yi);
                    }
                    double dE2 = A[2, j] * A[3, k] * (1 - A[3, k]) * acum;
                    Console.WriteLine("dError/dW(2)" + j.ToString() + "," + k.ToString() + " = " + dE2.ToString());
                    double nuevoPeso = W[2, j, k] - alpha * dE2;
                    Console.WriteLine("W(2)" + j.ToString() + "," + k.ToString() + " = " + nuevoPeso.ToString());
                }
```

La capa anterior

$$\frac{\partial Error}{\partial w_{j,k}^{(1)}} = x_j * a_k^{(2)} * \left(1 - a_k^{(2)}\right) * \sum_{p=1}^{n_3} \left[w_{k,p}^{(2)} * a_p^{(3)} * \left(1 - a_p^{(3)}\right) * \sum_{i=1}^{n_4} \left(w_{p,i}^{(3)} * (y_i - S_i) * y_i * (1 - y_i) \right) \right]$$

Y

$$w_{j,k}^{(1)} \leftarrow w_{j,k}^{(1)} - \propto * \frac{\partial Error}{\partial w_{j,k}^{(1)}}$$

Se implementa así:

```
            //Procesa capa 2
            for (int j = 1; j <= neuronasporcapa[1]; j++)
                for (int k = 1; k <= neuronasporcapa[2]; k++)
                {
                    double acumular = 0;
                    for (int p = 1; p <= neuronasporcapa[3]; p++)
                    {
                        double acum = 0;
                        for (int i = 1; i <= neuronasporcapa[4]; i++)
                        {
                            double Yi = A[4, i];
                            acum += W[3, p, i] * (Yi - S[i]) * Yi * (1 - Yi);
                        }
                        acumular += W[2, k, p] * A[3, p] * (1 - A[3, p]) * acum;
                    }
                    double dE1 = E[j] * A[2, k] * (1 - A[2, k]) * acumular;
                    Console.WriteLine("dError/dW(1)" + j.ToString() + "," + k.ToString() + " = " + dE1.ToString());
                    double nuevoPeso = W[1, j, k] - alpha * dE1;
                    Console.WriteLine("W(1)" + j.ToString() + "," + k.ToString() + " = " + nuevoPeso.ToString());
                }
```

Para los umbrales

$$\frac{\partial Error}{\partial u_i^{(4)}} = (y_i - S_i) * y_i * (1 - y_i)$$

$$u_i^{(4)} \leftarrow u_i^{(4)} - \propto * \frac{\partial Error}{\partial u_i^{(4)}}$$

```
//Ajusta umbrales capa 4
for (int i = 1; i <= neuronasporcapa[4]; i++)
{
    double Yi = A[4, i];
    double dE4 = (Yi - S[i]) * Yi * (1 - Yi);
    double nuevoUmbral = U[4, i] - alpha * dE4;
}
```

$$\frac{\partial Error}{\partial u_k^{(3)}} = a_k^{(3)} * \left(1 - a_k^{(3)}\right) * \sum_{i=1}^{n_4} \left(w_{k,i}^{(3)} * (y_i - S_i) * y_i * (1 - y_i)\right)$$

$$u_k^{(3)} \leftarrow u_k^{(3)} - \propto * \frac{\partial Error}{\partial u_k^{(3)}}$$

```
//Ajusta umbrales capa 3
for (int k = 1; k <= neuronasporcapa[3]; k++)
{
    double acum = 0;
    for (int i = 1; i <= neuronasporcapa[4]; i++)
    {
        double Yi = A[4, i];
        acum += W[3, k, i] * (Yi - S[i]) * Yi * (1 - Yi);
    }
    double dE3 = A[3, k] * (1 - A[3, k]) * acum;
    double nuevoUmbral = U[3, k] - alpha * dE3;
}
```

$$\frac{\partial Error}{\partial u_k^{(2)}} = a_k^{(2)} * \left(1 - a_k^{(2)}\right) * \sum_{p=1}^{n_3} \left[w_{k,p}^{(2)} * a_p^{(3)} * \left(1 - a_p^{(3)}\right) * \sum_{i=1}^{n_4} \left(w_{p,i}^{(3)} * (y_i - S_i) * y_i * (1 - y_i)\right) \right]$$

$$u_k^{(2)} \leftarrow u_k^{(2)} - \propto * \frac{\partial Error}{\partial u_k^{(2)}}$$

```
//Ajusta umbrales capa 2
for (int k = 1; k <= neuronasporcapa[2]; k++)
{
    double acumular = 0;
    for (int p = 1; p <= neuronasporcapa[3]; p++)
    {
        double acum = 0;
        for (int i = 1; i <= neuronasporcapa[4]; i++)
        {
            double Yi = A[4, i];
            acum += W[3, p, i] * (Yi - S[i]) * Yi * (1 - Yi);
        }
        acumular += W[2, k, p] * A[3, p] * (1 - A[3, p]) * acum;
    }
    double dE2 = A[2, k] * (1 - A[2, k]) * acumular;
    double nuevoUmbral = U[2, k] - alpha * dE2;
}
```

Código del perceptrón en una clase implementado en C#

A continuación se muestra el código completo en el que se ha creado una clase que implementa el perceptrón (creación, proceso, entrenamiento) y en la clase principal se pone como datos de prueba la tabla del XOR que el perceptrón debe aprender.

```
using System;

namespace RedNeuronal3 {

    //La clase que implementa el perceptrón
    class Perceptron {
        private double[,,] W; //Los pesos serán arreglos multidimensionales. Así: W[capa, neurona inicial, neurona final]
        private double[,] U; //Los umbrales de cada neurona serán arreglos bidimensionales. Así: U[capa, neurona que produce la salida]
        double[,] A; //Las salidas de cada neurona serán arreglos bidimensionales. Así: A[capa, neurona que produce la salida]

        private double[,,] WN; //Los nuevos pesos serán arreglos multidimensionales. Así: W[capa, neurona inicial, neurona final]
        private double[,] UN; //Los nuevos umbrales de cada neurona serán arreglos bidimensionales. Así: U[capa, neurona que produce la salida]

        private int TotalCapas; //El total de capas que tendrá el perceptrón incluyendo la capa de entrada
        private int[] neuronasporcapa; //Cuantas neuronas habrá en cada capa
        private int TotalEntradas; //Total de entradas externas del perceptrón
        private int TotalSalidas; //Total salidas externas del perceptrón

        public Perceptron(int TotalEntradas, int TotalSalidas, int TotalCapas, int[] neuronasporcapa) {
            this.TotalEntradas = TotalEntradas;
            this.TotalSalidas = TotalSalidas;
            this.TotalCapas = TotalCapas;
            int maxNeuronas = 0; //Detecta el máximo número de neuronas por capa para dimensionar los arreglos
            this.neuronasporcapa = new int[TotalCapas + 1];
            for (int capa = 1; capa <= TotalCapas; capa++) {
                this.neuronasporcapa[capa] = neuronasporcapa[capa];
                if (neuronasporcapa[capa] > maxNeuronas) maxNeuronas = neuronasporcapa[capa];
            }

            //Dimensiona con el máximo valor
            W = new double[TotalCapas + 1, maxNeuronas + 1, maxNeuronas + 1];
            U = new double[TotalCapas + 1, maxNeuronas + 1];
            WN = new double[TotalCapas + 1, maxNeuronas + 1, maxNeuronas + 1];
            UN = new double[TotalCapas + 1, maxNeuronas + 1];
            A = new double[TotalCapas + 1, maxNeuronas + 1];

            //Da valores aleatorios a pesos y umbrales
            Random azar = new Random();

            for (int capa = 2; capa <= TotalCapas; capa++)
                for (int i = 1; i <= neuronasporcapa[capa]; i++)
                    U[capa, i] = azar.NextDouble();

            for (int capa = 1; capa < TotalCapas; capa++)
                for (int i = 1; i <= neuronasporcapa[capa]; i++)
                    for (int j = 1; j <= neuronasporcapa[capa+1]; j++)
                        W[capa, i, j] = azar.NextDouble();
        }

        public void Procesa(double[] E) {
            //Entradas externas del perceptrón pasan a la salida de la primera capa
            for (int copia = 1; copia <= TotalEntradas; copia++) A[1, copia] = E[copia];

            //Proceso del perceptrón
            for (int capa = 2; capa <= TotalCapas; capa++)
                for (int neurona = 1; neurona <= neuronasporcapa[capa]; neurona++) {
                    A[capa, neurona] = 0;
                    for (int entra = 1; entra <= neuronasporcapa[capa - 1]; entra++)
                        A[capa, neurona] += A[capa - 1, entra] * W[capa - 1, entra, neurona];
                    A[capa, neurona] += U[capa, neurona];
                    A[capa, neurona] = 1 / (1 + Math.Exp(-A[capa, neurona]));
                }
        }

        // Muestra las entradas externas del perceptrón, las salidas esperadas y las salidas reales
        public void Muestra(double[] E, double[] S) {
            for (int cont = 1; cont <= TotalEntradas; cont++) Console.Write(E[cont] + ","); Console.Write(" = ");
            for (int cont = 1; cont <= TotalSalidas; cont++) Console.Write(S[cont] + ","); Console.Write(" <vs> ");
            for (int cont = 1; cont <= TotalSalidas; cont++) //Salidas reales del perceptrón
                if (A[TotalCapas, cont] > 0.5) //El umbral: Mayor de 0.5 es 1, de lo contrario es cero
                    Console.Write("1,  " + A[TotalCapas, cont]); //Salida binaria y salida real
                else
                    Console.Write("0,  " + A[TotalCapas, cont]);
            Console.WriteLine(" ");
        }

        //El entrenamiento es ajustar los pesos y umbrales
        public void Entrena(double alpha, double[] E, double[] S) {
            //Ajusta pesos capa3 ==> capa4
            for (int j = 1; j <= neuronasporcapa[3]; j++)
                for (int i = 1; i <= neuronasporcapa[4]; i++) {
                    double Yi = A[4, i];
                    double dE3 = A[3, j] * (Yi - S[i]) * Yi * (1 - Yi);
                    WN[3, j, i] = W[3, j, i] - alpha * dE3; //Nuevo peso se guarda temporalmente
                }

            //Ajusta pesos capa2 ==> capa3
```

```
            for (int j = 1; j <= neuronasporcapa[2]; j++)
                for (int k = 1; k <= neuronasporcapa[3]; k++) {
                    double acum = 0;
                    for (int i = 1; i <= neuronasporcapa[4]; i++) {
                        double Yi = A[4, i];
                        acum += W[3, k, i] * (Yi - S[i]) * Yi * (1 - Yi);
                    }
                    double dE2 = A[2, j] * A[3, k] * (1 - A[3, k]) * acum;
                    WN[2, j, k] = W[2, j, k] - alpha * dE2; //Nuevo peso se guarda temporalmente
                }

            //Ajusta pesos capa1 ==> capa2
            for (int j = 1; j <= neuronasporcapa[1]; j++)
                for (int k = 1; k <= neuronasporcapa[2]; k++) {
                    double acumular = 0;
                    for (int p = 1; p <= neuronasporcapa[3]; p++) {
                        double acum = 0;
                        for (int i = 1; i <= neuronasporcapa[4]; i++) {
                            double Yi = A[4, i];
                            acum += W[3, p, i] * (Yi - S[i]) * Yi * (1 - Yi);
                        }
                        acumular += W[2, k, p] * A[3, p] * (1 - A[3, p]) * acum;
                    }
                    double dE1 = E[j] * A[2, k] * (1 - A[2, k]) * acumular;
                    WN[1, j, k] = W[1, j, k] - alpha * dE1; //Nuevo peso se guarda temporalmente
                }

            //Ajusta umbrales de neuronas de la capa 4
            for (int i = 1; i <= neuronasporcapa[4]; i++) {
                double Yi = A[4, i];
                double dE4 = (Yi - S[i]) * Yi * (1 - Yi);
                UN[4, i] = U[4, i] - alpha * dE4; //Nuevo umbral se guarda temporalmente
            }

            //Ajusta umbrales de neuronas de la capa 3
            for (int k = 1; k <= neuronasporcapa[3]; k++) {
                double acum = 0;
                for (int i = 1; i <= neuronasporcapa[4]; i++) {
                    double Yi = A[4, i];
                    acum += W[3, k, i] * (Yi - S[i]) * Yi * (1 - Yi);
                }
                double dE3 = A[3, k] * (1 - A[3, k]) * acum;
                UN[3, k] = U[3, k] - alpha * dE3; //Nuevo umbral se guarda temporalmente
            }

            //Ajusta umbrales de neuronas de la capa 2
            for (int k = 1; k <= neuronasporcapa[2]; k++) {
                double acumular = 0;
                for (int p = 1; p <= neuronasporcapa[3]; p++) {
                    double acum = 0;
                    for (int i = 1; i <= neuronasporcapa[4]; i++) {
                        double Yi = A[4, i];
                        acum += W[3, p, i] * (Yi - S[i]) * Yi * (1 - Yi);
                    }
                    acumular += W[2, k, p] * A[3, p] * (1 - A[3, p]) * acum;
                }
                double dE2 = A[2, k] * (1 - A[2, k]) * acumular;
                UN[2, k] = U[2, k] - alpha * dE2; //Nuevo umbral se guarda temporalmente
            }


            //Copia los nuevos pesos y umbrales a los pesos y umbrales respectivos del perceptrón
            for (int capa = 2; capa <= TotalCapas; capa++)
                for (int i = 1; i <= neuronasporcapa[capa]; i++)
                    U[capa, i] = UN[capa, i];

            for (int capa = 1; capa < TotalCapas; capa++)
                for (int i = 1; i <= neuronasporcapa[capa]; i++)
                    for (int j = 1; j <= neuronasporcapa[capa + 1]; j++)
                        W[capa, i, j] = WN[capa, i, j];
        }
    }

    class Program {
        static void Main(string[] args)
        {
            int TotalEntradas = 2; //Número de entradas externas del perceptrón
            int TotalSalidas = 1; //Número de salidas externas del perceptrón
            int TotalCapas = 4; //Total capas que tendrá el perceptrón
            int[] neuronasporcapa = new int[TotalCapas + 1]; //Los índices iniciarán en 1 en esta implementación
            neuronasporcapa[1] = TotalEntradas; //Entradas externas del perceptrón
            neuronasporcapa[2] = 4; //Capa oculta con 4 neuronas
            neuronasporcapa[3] = 4; //Capa oculta con 4 neuronas
            neuronasporcapa[4] = TotalSalidas; //Capa de salida con 2 neuronas

            Perceptron objP = new Perceptron(TotalEntradas, TotalSalidas, TotalCapas, neuronasporcapa);

            /* Tabla del XOR. Son 4 conjuntos de entradas y salidas
                1 ..... 1  ===> 0
                1 ..... 0  ===> 1
                0 ..... 1  ===> 1
                0 ..... 0  ===> 0     */
            int ConjuntoEntradas = 4;
            double[][] entraXOR = new double[ConjuntoEntradas+1][];
            entraXOR[1] = new double[3];
            entraXOR[2] = new double[3];
            entraXOR[3] = new double[3];
            entraXOR[4] = new double[3];
            entraXOR[1][1] = 1; entraXOR[2][1] = 1; entraXOR[3][1] = 0; entraXOR[4][1] = 0;
```

```
            entraXOR[1][2] = 1; entraXOR[2][2] = 0; entraXOR[3][2] = 1; entraXOR[4][2] = 0;

            double[][] salirXOR = new double[ConjuntoEntradas+1][];
            salirXOR[1] = new double[3];
            salirXOR[2] = new double[3];
            salirXOR[3] = new double[3];
            salirXOR[4] = new double[3];
            salirXOR[1][1] = 0; salirXOR[2][1] = 1; salirXOR[3][1] = 1; salirXOR[4][1] = 0;

            double alpha = 0.4; //Factor de aprendizaje

            //Ciclo que entrena la red neuronal
            for (int ciclo = 1; ciclo <= 8000; ciclo++) {
                if (ciclo % 500 == 0) Console.WriteLine("Iteracion: " + ciclo);
                //Importante: Se envía el primer conjunto de entradas-salidas, luego el segundo, tercero y cuarto
                //por cada ciclo de entrenamiento.
                for (int entra = 1; entra <= ConjuntoEntradas; entra++) {
                    objP.Procesa(entraXOR[entra]);
                    if (ciclo % 500 == 0) objP.Muestra(entraXOR[entra], salirXOR[entra]);
                    objP.Entrena(alpha, entraXOR[entra], salirXOR[entra]);
                }
            }

            Console.ReadKey();
        }
    }
}
```

Ejemplo de ejecución

```
file:///C:/Users/engin/OneDrive/Documentos/Visual Studio 2015/Projects/RedNeuronal3/RedNeuronal3/bin/Debug/RedNeuronal3.EXE
Iteracion: 500
1,1, = 0, <vs> 1,  0,501368977621328
1,0, = 1, <vs> 0,  0,468664650629771
0,1, = 1, <vs> 1,  0,501195550445656
0,0, = 0, <vs> 1,  0,529424467005851
Iteracion: 1000
1,1, = 0, <vs> 1,  0,500619150537241
1,0, = 1, <vs> 0,  0,477788581256291
0,1, = 1, <vs> 1,  0,500252266838435
0,0, = 0, <vs> 1,  0,520083856761412
Iteracion: 1500
1,1, = 0, <vs> 0,  0,499976235615496
1,0, = 1, <vs> 0,  0,48222873747768
0,1, = 1, <vs> 1,  0,500033719491311
0,0, = 0, <vs> 1,  0,516474130070833
Iteracion: 2000
1,1, = 0, <vs> 1,  0,500236498154176
1,0, = 1, <vs> 0,  0,484192922453521
0,1, = 1, <vs> 1,  0,500041081675172
0,0, = 0, <vs> 1,  0,514461996246876
Iteracion: 2500
1,1, = 0, <vs> 1,  0,501060813275512
1,0, = 1, <vs> 0,  0,485199152224443
0,1, = 1, <vs> 1,  0,500090147188155
0,0, = 0, <vs> 1,  0,512648417980808
Iteracion: 3000
1,1, = 0, <vs> 1,  0,502467604076654
1,0, = 1, <vs> 0,  0,485864592786955
0,1, = 1, <vs> 1,  0,500180334862566
0,0, = 0, <vs> 1,  0,510368291859718
```

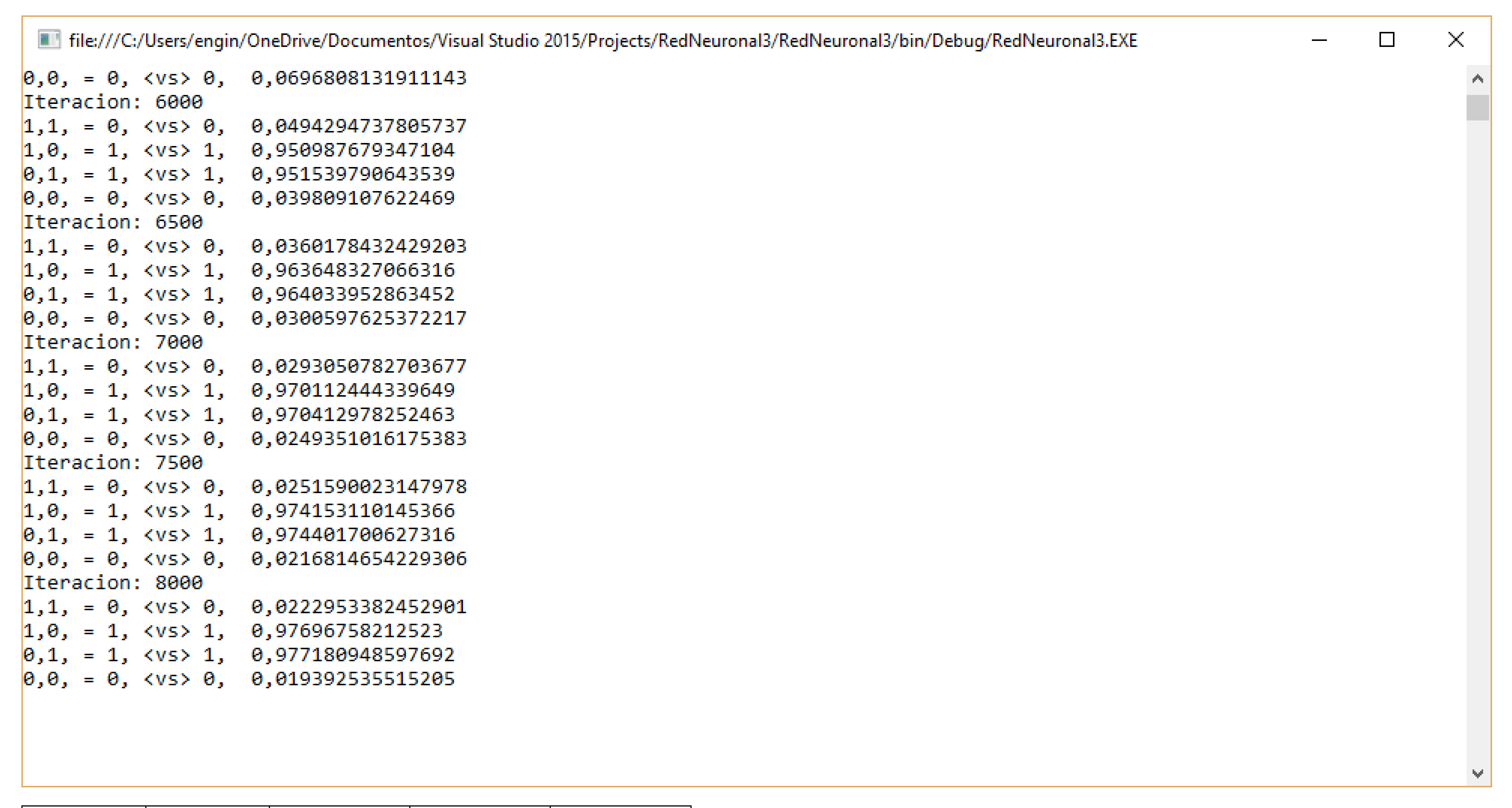
El programa muestra la tabla del XOR que se debe aprender

Valor	Valor	Resultado esperado
1	1	0
1	0	1
0	1	1
0	0	0

Luego imprime "<vs>", el valor aprendido en esa iteración (un valor de 1 o 0) y el valor del número real de la salida. Obsérvese que en la iteración 2000 se obtuvo

Valor	Valor	Resultado esperado	Resultado Real	
1	1	0	1	MAL
1	0	1	0	MAL
0	1	1	1	BIEN
0	0	0	1	MAL

Más adelante en la iteración 8000 se obtuvo



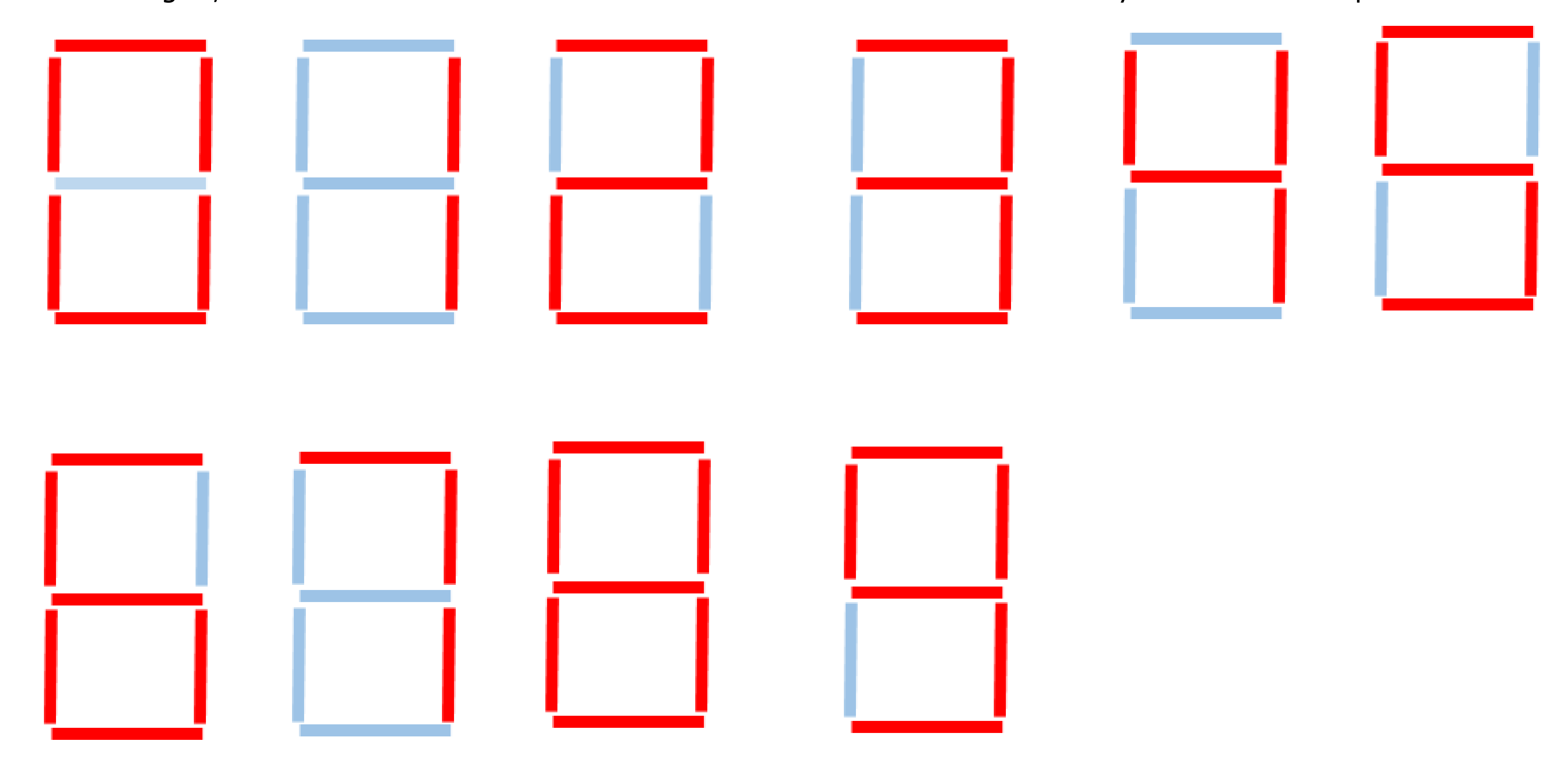
```
file:///C:/Users/engin/OneDrive/Documentos/Visual Studio 2015/Projects/RedNeuronal3/RedNeuronal3/bin/Debug/RedNeuronal3.EXE
0,0, = 0, <vs> 0,  0,0696808131911143
Iteracion: 6000
1,1, = 0, <vs> 0,  0,0494294737805737
1,0, = 1, <vs> 1,  0,950987679347104
0,1, = 1, <vs> 1,  0,951539790643539
0,0, = 0, <vs> 0,  0,039809107622469
Iteracion: 6500
1,1, = 0, <vs> 0,  0,0360178432429203
1,0, = 1, <vs> 1,  0,963648327066316
0,1, = 1, <vs> 1,  0,964033952863452
0,0, = 0, <vs> 0,  0,0300597625372217
Iteracion: 7000
1,1, = 0, <vs> 0,  0,0293050782703677
1,0, = 1, <vs> 1,  0,970112444339649
0,1, = 1, <vs> 1,  0,970412978252463
0,0, = 0, <vs> 0,  0,0249351016175383
Iteracion: 7500
1,1, = 0, <vs> 0,  0,0251590023147978
1,0, = 1, <vs> 1,  0,974153110145366
0,1, = 1, <vs> 1,  0,974401700627316
0,0, = 0, <vs> 0,  0,0216814654229306
Iteracion: 8000
1,1, = 0, <vs> 0,  0,0222953382452901
1,0, = 1, <vs> 1,  0,97696758212523
0,1, = 1, <vs> 1,  0,977180948597692
0,0, = 0, <vs> 0,  0,019392535515205
```

Valor	Valor	Resultado esperado	Resultado Real	
1	1	0	0	BIEN
1	0	1	1	BIEN
0	1	1	1	BIEN
0	0	0	0	BIEN

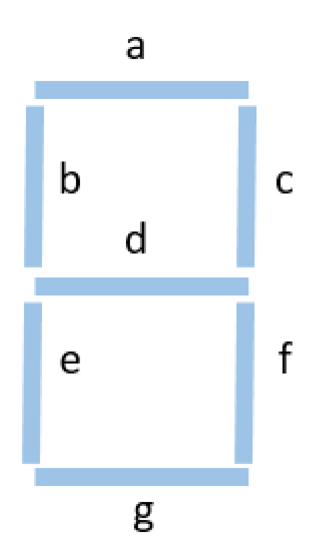
En la última columna se muestra el valor real (por ejemplo, 0.019392535515205) que tiene el perceptrón. Esos valores se ajustan a 0 o a 1 poniendo el umbral en 0.5, si es mayor de 0.5 entonces se toma como 1 en caso contrario sería 0. Importante: Ese ajuste a 0 o a 1 es para mostrarlo al usuario final, el valor real debe conservarse para calcular el error y hacer uso del algoritmo de "backpropagation"

Reconocimiento de números de un reloj digital

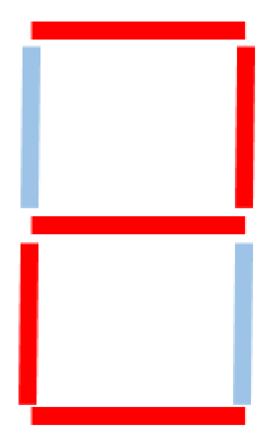
En la imagen, los números del 0 al 9 construidos usando las barras verticales y horizontales. Típicos de un reloj digital.



Se quiere construir una red neuronal tipo perceptrón multicapa que dado ese número al estilo reloj digital se pueda deducir el número como tal. Para iniciar se pone un identificador a cada barra



Luego se le da un valor de "1" a la barra que queda en rojo al construir el número y "0" a la barra que queda en azul claro. Por ejemplo:



Los valores de a,b,c,d,e,f,g serían: 1,0,1,1,1,0,1

Como se debe deducir que es un 2, este valor se convierte a binario 10_2 o 0,0,1,0 (para convertirlo en salida del perceptrón, como el máximo valor es 9 y este es 1,0,0,1 entonces el número de salidas es 4)

La tabla de entradas y salidas esperadas es:

Imagen	Valor de entrada	Valor de salida esperado
	1,1,1,0,1,1,1	0,0,0,0
	0,0,1,0,0,1,0	0,0,0,1
	1,0,1,1,1,0,1	0,0,1,0
	1,0,1,1,0,1,1	0,0,1,1
	0,1,1,1,0,1,0	0,1,0,0
	1,1,0,1,0,1,1	0,1,0,1
	1,1,0,1,1,1,1	0,1,1,0

	1,0,1,0,0,1,0	0,1,1,1
	1,1,1,1,1,1,1	1,0,0,0
	1,1,1,1,0,1,1	1,0,0,1

El código es el siguiente

```
using System;

namespace NeuroNumero {

    //La clase que implementa el perceptrón
    class Perceptron {
        private double[,,] W; //Los pesos serán arreglos multidimensionales. Así: W[capa, neurona inicial, neurona final]
        private double[,] U; //Los umbrales de cada neurona serán arreglos bidimensionales. Así: U[capa, neurona que produce la salida]
        double[,] A; //Las salidas de cada neurona serán arreglos bidimensionales. Así: A[capa, neurona que produce la salida]

        private double[,,] WN; //Los nuevos pesos serán arreglos multidimensionales. Así: W[capa, neurona inicial, neurona final]
        private double[,] UN; //Los nuevos umbrales de cada neurona serán arreglos bidimensionales. Así: U[capa, neurona que produce la salida]

        private int TotalCapas; //El total de capas que tendrá el perceptrón incluyendo la capa de entrada
        private int[] neuronasporcapa; //Cuantas neuronas habrá en cada capa
        private int TotalEntradas; //Total de entradas externas del perceptrón
        private int TotalSalidas; //Total salidas externas del perceptrón

        public Perceptron(int TotalEntradas, int TotalSalidas, int TotalCapas, int[] neuronasporcapa) {
            this.TotalEntradas = TotalEntradas;
            this.TotalSalidas = TotalSalidas;
            this.TotalCapas = TotalCapas;
            int maxNeuronas = 0; //Detecta el máximo número de neuronas por capa para dimensionar los arreglos
            this.neuronasporcapa = new int[TotalCapas + 1];
            for (int capa = 1; capa <= TotalCapas; capa++) {
                this.neuronasporcapa[capa] = neuronasporcapa[capa];
                if (neuronasporcapa[capa] > maxNeuronas) maxNeuronas = neuronasporcapa[capa];
            }

            //Dimensiona con el máximo valor
            W = new double[TotalCapas + 1, maxNeuronas + 1, maxNeuronas + 1];
            U = new double[TotalCapas + 1, maxNeuronas + 1];
            WN = new double[TotalCapas + 1, maxNeuronas + 1, maxNeuronas + 1];
            UN = new double[TotalCapas + 1, maxNeuronas + 1];
            A = new double[TotalCapas + 1, maxNeuronas + 1];

            //Da valores aleatorios a pesos y umbrales
            Random azar = new Random();

            for (int capa = 2; capa <= TotalCapas; capa++)
                for (int i = 1; i <= neuronasporcapa[capa]; i++)
                    U[capa, i] = azar.NextDouble();

            for (int capa = 1; capa < TotalCapas; capa++)
                for (int i = 1; i <= neuronasporcapa[capa]; i++)
                    for (int j = 1; j <= neuronasporcapa[capa+1]; j++)
                        W[capa, i, j] = azar.NextDouble();
        }

        public void Procesa(double[] E) {
            //Entradas externas del perceptrón pasan a la salida de la primera capa
            for (int copia = 1; copia <= TotalEntradas; copia++) A[1, copia] = E[copia];

            //Proceso del perceptrón
            for (int capa = 2; capa <= TotalCapas; capa++)
                for (int neurona = 1; neurona <= neuronasporcapa[capa]; neurona++) {
                    A[capa, neurona] = 0;
                    for (int entra = 1; entra <= neuronasporcapa[capa - 1]; entra++)
                        A[capa, neurona] += A[capa - 1, entra] * W[capa - 1, entra, neurona];
                    A[capa, neurona] += U[capa, neurona];
                    A[capa, neurona] = 1 / (1 + Math.Exp(-A[capa, neurona]));
                }
        }

        // Muestra las entradas externas del perceptrón, las salidas esperadas y las salidas reales
        public void Muestra(double[] E, double[] S) {
            for (int cont = 1; cont <= TotalEntradas; cont++) Console.Write(E[cont] + ","); Console.Write(" = ");
            for (int cont = 1; cont <= TotalSalidas; cont++) Console.Write(S[cont] + ","); Console.Write(" <vs> ");
            for (int cont = 1; cont <= TotalSalidas; cont++) //Salidas reales del perceptrón
                if (A[TotalCapas, cont] > 0.5) //El umbral: Mayor de 0.5 es 1, de lo contrario es cero
                    Console.Write("1,");
                else
                    Console.Write("0,");
            Console.WriteLine(" ");
        }

        //El entrenamiento es ajustar los pesos y umbrales
        public void Entrena(double alpha, double[] E, double[] S) {
            //Ajusta pesos capa3 ==> capa4
            for (int j = 1; j <= neuronasporcapa[3]; j++)
                for (int i = 1; i <= neuronasporcapa[4]; i++) {
                    double Yi = A[4, i];
                    double dE3 = A[3, j] * (Yi - S[i]) * Yi * (1 - Yi);
                    WN[3, j, i] = W[3, j, i] - alpha * dE3; //Nuevo peso se guarda temporalmente
                }

            //Ajusta pesos capa2 ==> capa3
            for (int j = 1; j <= neuronasporcapa[2]; j++)
                for (int k = 1; k <= neuronasporcapa[3]; k++) {
                    double acum = 0;
                    for (int i = 1; i <= neuronasporcapa[4]; i++) {
                        double Yi = A[4, i];
                        acum += W[3, k, i] * (Yi - S[i]) * Yi * (1 - Yi);
                    }
                    double dE2 = A[2, j] * A[3, k] * (1 - A[3, k]) * acum;
                    WN[2, j, k] = W[2, j, k] - alpha * dE2; //Nuevo peso se guarda temporalmente
```

```
            }

        //Ajusta pesos capa1 ==> capa2
        for (int j = 1; j <= neuronasporcapa[1]; j++)
            for (int k = 1; k <= neuronasporcapa[2]; k++) {
                double acumular = 0;
                for (int p = 1; p <= neuronasporcapa[3]; p++) {
                    double acum = 0;
                    for (int i = 1; i <= neuronasporcapa[4]; i++) {
                        double Yi = A[4, i];
                        acum += W[3, p, i] * (Yi - S[i]) * Yi * (1 - Yi);
                    }
                    acumular += W[2, k, p] * A[3, p] * (1 - A[3, p]) * acum;
                }
                double dE1 = E[j] * A[2, k] * (1 - A[2, k]) * acumular;
                WN[1, j, k] = W[1, j, k] - alpha * dE1; //Nuevo peso se guarda temporalmente
            }

        //Ajusta umbrales de neuronas de la capa 4
        for (int i = 1; i <= neuronasporcapa[4]; i++) {
            double Yi = A[4, i];
            double dE4 = (Yi - S[i]) * Yi * (1 - Yi);
            UN[4, i] = U[4, i] - alpha * dE4; //Nuevo umbral se guarda temporalmente
        }

        //Ajusta umbrales de neuronas de la capa 3
        for (int k = 1; k <= neuronasporcapa[3]; k++) {
            double acum = 0;
            for (int i = 1; i <= neuronasporcapa[4]; i++)
            {
                double Yi = A[4, i];
                acum += W[3, k, i] * (Yi - S[i]) * Yi * (1 - Yi);
            }
            double dE3 = A[3, k] * (1 - A[3, k]) * acum;
            UN[3, k] = U[3, k] - alpha * dE3; //Nuevo umbral se guarda temporalmente
        }

        //Ajusta umbrales de neuronas de la capa 2
        for (int k = 1; k <= neuronasporcapa[2]; k++) {
            double acumular = 0;
            for (int p = 1; p <= neuronasporcapa[3]; p++) {
                double acum = 0;
                for (int i = 1; i <= neuronasporcapa[4]; i++) {
                    double Yi = A[4, i];
                    acum += W[3, p, i] * (Yi - S[i]) * Yi * (1 - Yi);
                }
                acumular += W[2, k, p] * A[3, p] * (1 - A[3, p]) * acum;
            }
            double dE2 = A[2, k] * (1 - A[2, k]) * acumular;
            UN[2, k] = U[2, k] - alpha * dE2; //Nuevo umbral se guarda temporalmente
        }

        //Copia los nuevos pesos y umbrales a los pesos y umbrales respectivos del perceptrón
        for (int capa = 2; capa <= TotalCapas; capa++)
            for (int i = 1; i <= neuronasporcapa[capa]; i++)
                U[capa, i] = UN[capa, i];

        for (int capa = 1; capa < TotalCapas; capa++)
            for (int i = 1; i <= neuronasporcapa[capa]; i++)
                for (int j = 1; j <= neuronasporcapa[capa + 1]; j++)
                    W[capa, i, j] = WN[capa, i, j];
    }
}

class Program {
    static void Main(string[] args)
    {
        int TotalEntradas = 7; //Número de entradas externas del perceptrón
        int TotalSalidas = 4; //Número de salidas externas del perceptrón
        int TotalCapas = 4; //Total capas que tendrá el perceptrón
        int[] neuronasporcapa = new int[TotalCapas + 1]; //Los índices iniciarán en 1 en esta implementación
        neuronasporcapa[1] = TotalEntradas; //Entradas externas del perceptrón
        neuronasporcapa[2] = 4; //Capa oculta con 4 neuronas
        neuronasporcapa[3] = 4; //Capa oculta con 4 neuronas
        neuronasporcapa[4] = TotalSalidas; //Capa de salida con 2 neuronas

        Perceptron objP = new Perceptron(TotalEntradas, TotalSalidas, TotalCapas, neuronasporcapa);

        /*
            Entero     Binario       Figura
            0       0000      1110111
            1       0001      0010010
            2       0010      1011101
            3       0011      1011011
            4       0100      0111010
            5       0101      1101011
            6       0110      1101111
            7       0111      1010010
            8       1000      1111111
            9       1001      1111011
        */
        int ConjuntoEntradas = 10;
        double[][] entraFigura = new double[ConjuntoEntradas + 1][];
        entraFigura[1] = new double[] { 0, 1, 1, 1, 0, 1, 1, 1 };
        entraFigura[2] = new double[] { 0, 0, 0, 1, 0, 0, 1, 0 };
        entraFigura[3] = new double[] { 0, 1, 0, 1, 1, 1, 0, 1 };
        entraFigura[4] = new double[] { 0, 1, 0, 1, 1, 0, 1, 1 };
        entraFigura[5] = new double[] { 0, 0, 1, 1, 1, 0, 1, 0 };
        entraFigura[6] = new double[] { 0, 1, 1, 0, 1, 0, 1, 1 };
```

```
            entraFigura[7] = new double[] { 0, 1, 1, 0, 1, 1, 1, 1 };
            entraFigura[8] = new double[] { 0, 1, 0, 1, 0, 0, 1, 0 };
            entraFigura[9] = new double[] { 0, 1, 1, 1, 1, 1, 1, 1 };
            entraFigura[10] = new double[] { 0, 1, 1, 1, 1, 0, 1, 1 };

            double[][] saleBinario = new double[ConjuntoEntradas + 1][];
            saleBinario[1] = new double[] { 0, 0, 0, 0, 0 };
            saleBinario[2] = new double[] { 0, 0, 0, 0, 1 };
            saleBinario[3] = new double[] { 0, 0, 0, 1, 0 };
            saleBinario[4] = new double[] { 0, 0, 0, 1, 1 };
            saleBinario[5] = new double[] { 0, 0, 1, 0, 0 };
            saleBinario[6] = new double[] { 0, 0, 1, 0, 1 };
            saleBinario[7] = new double[] { 0, 0, 1, 1, 0 };
            saleBinario[8] = new double[] { 0, 0, 1, 1, 1 };
            saleBinario[9] = new double[] { 0, 1, 0, 0, 0 };
            saleBinario[10] = new double[] { 0, 1, 0, 0, 1 };

            double alpha = 0.4; //Factor de aprendizaje
            for (int ciclo = 1; ciclo <= 8000; ciclo++) {
                if (ciclo % 500 == 0) Console.WriteLine("Iteracion: " + ciclo);
                //Importante: Se envía el primer conjunto de entradas-salidas, luego el segundo, tercero y cuarto
                //por cada ciclo de entrenamiento.
                for (int entra = 1; entra <= ConjuntoEntradas; entra++) {
                    objP.Procesa(entraFigura[entra]);
                    if (ciclo % 500 == 0) objP.Muestra(entraFigura[entra], saleBinario[entra]);
                    objP.Entrena(alpha, entraFigura[entra], saleBinario[entra]);
                }
            }

            Console.ReadKey();
        }
    }
}
```

La clase Perceptron no ha variado, sigue siendo la misma que la usada para aprender la tabla XOR (ejemplo anterior). Los cambios se dan en la clase principal en los cuales se cambia el número de entradas y salidas, y por supuesto, los valores de entrada y las salidas esperadas.

Esta sería su ejecución

```
file:///C:/Users/engin/onedrive/documentos/visual studio 2015/Projects/NeuroNumero/NeuroNumero/bin/Debug/NeuroNumero.EXE
Iteracion: 500
1,1,1,0,1,1,1, = 0,0,0,0, <vs> 0,0,0,0,
0,0,1,0,0,1,0, = 0,0,0,1, <vs> 0,0,0,1,
1,0,1,1,1,0,1, = 0,0,1,0, <vs> 0,0,0,0,
1,0,1,1,0,1,1, = 0,0,1,1, <vs> 0,0,1,1,
0,1,1,1,0,1,0, = 0,1,0,0, <vs> 0,0,0,0,
1,1,0,1,0,1,1, = 0,1,0,1, <vs> 0,0,0,1,
1,1,0,1,1,1,1, = 0,1,1,0, <vs> 0,0,0,0,
1,0,1,0,0,1,0, = 0,1,1,1, <vs> 0,0,1,1,
1,1,1,1,1,1,1, = 1,0,0,0, <vs> 0,0,0,0,
1,1,1,1,0,1,1, = 1,0,0,1, <vs> 0,0,0,1,
Iteracion: 1000
1,1,1,0,1,1,1, = 0,0,0,0, <vs> 0,0,0,0,
0,0,1,0,0,1,0, = 0,0,0,1, <vs> 0,0,0,1,
1,0,1,1,1,0,1, = 0,0,1,0, <vs> 0,0,1,0,
1,0,1,1,0,1,1, = 0,0,1,1, <vs> 0,0,1,1,
0,1,1,1,0,1,0, = 0,1,0,0, <vs> 0,1,0,0,
1,1,0,1,0,1,1, = 0,1,0,1, <vs> 0,1,0,1,
1,1,0,1,1,1,1, = 0,1,1,0, <vs> 0,1,0,0,
1,0,1,0,0,1,0, = 0,1,1,1, <vs> 0,1,1,1,
1,1,1,1,1,1,1, = 1,0,0,0, <vs> 0,0,0,0,
1,1,1,1,0,1,1, = 1,0,0,1, <vs> 0,0,0,1,
Iteracion: 1500
1,1,1,0,1,1,1, = 0,0,0,0, <vs> 0,0,0,0,
0,0,1,0,0,1,0, = 0,0,0,1, <vs> 0,0,0,1,
1,0,1,1,1,0,1, = 0,0,1,0, <vs> 0,0,1,0,
1,0,1,1,0,1,1, = 0,0,1,1, <vs> 0,0,1,1,
0,1,1,1,0,1,0, = 0,1,0,0, <vs> 0,1,0,0,
1,1,0,1,0,1,1, = 0,1,0,1, <vs> 0,1,0,1,
1,1,0,1,1,1,1, = 0,1,1,0, <vs> 0,1,0,0,
```

En la iteración 500, comienza a notarse que la red neuronal está aprendiendo el patrón. Requiere más iteraciones

file:///C:/Users/engin/onedrive/documentos/visual studio 2015/Projects/NeuroNumero/NeuroNumero/bin/Debug/NeuroNumero.EXE

```
1,1,0,1,0,1,1, = 0,1,0,1, <vs> 0,1,0,1,
1,1,0,1,1,1,1, = 0,1,1,0, <vs> 0,1,1,0,
1,0,1,0,0,1,0, = 0,1,1,1, <vs> 0,1,1,1,
1,1,1,1,1,1,1, = 1,0,0,0, <vs> 1,0,0,0,
1,1,1,1,0,1,1, = 1,0,0,1, <vs> 1,0,0,1,
Iteracion: 7500
1,1,1,0,1,1,1, = 0,0,0,0, <vs> 0,0,0,0,
0,0,1,0,0,1,0, = 0,0,0,1, <vs> 0,0,0,1,
1,0,1,1,1,0,1, = 0,0,1,0, <vs> 0,0,1,0,
1,0,1,1,0,1,1, = 0,0,1,1, <vs> 0,0,1,1,
0,1,1,1,0,1,0, = 0,1,0,0, <vs> 0,1,0,0,
1,1,0,1,0,1,1, = 0,1,0,1, <vs> 0,1,0,1,
1,1,0,1,1,1,1, = 0,1,1,0, <vs> 0,1,1,0,
1,0,1,0,0,1,0, = 0,1,1,1, <vs> 0,1,1,1,
1,1,1,1,1,1,1, = 1,0,0,0, <vs> 1,0,0,0,
1,1,1,1,0,1,1, = 1,0,0,1, <vs> 1,0,0,1,
Iteracion: 8000
1,1,1,0,1,1,1, = 0,0,0,0, <vs> 0,0,0,0,
0,0,1,0,0,1,0, = 0,0,0,1, <vs> 0,0,0,1,
1,0,1,1,1,0,1, = 0,0,1,0, <vs> 0,0,1,0,
1,0,1,1,0,1,1, = 0,0,1,1, <vs> 0,0,1,1,
0,1,1,1,0,1,0, = 0,1,0,0, <vs> 0,1,0,0,
1,1,0,1,0,1,1, = 0,1,0,1, <vs> 0,1,0,1,
1,1,0,1,1,1,1, = 0,1,1,0, <vs> 0,1,1,0,
1,0,1,0,0,1,0, = 0,1,1,1, <vs> 0,1,1,1,
1,1,1,1,1,1,1, = 1,0,0,0, <vs> 1,0,0,0,
1,1,1,1,0,1,1, = 1,0,0,1, <vs> 1,0,0,1,
```

En la iteración 7500 ya se observa que la red está entrenada completamente para reconocer los números. Solo faltaría implementar que se detenga el entrenamiento cuando la red neuronal ofrezca las salidas esperadas y así evitar iteraciones de más.

Detección de patrones en series de tiempo

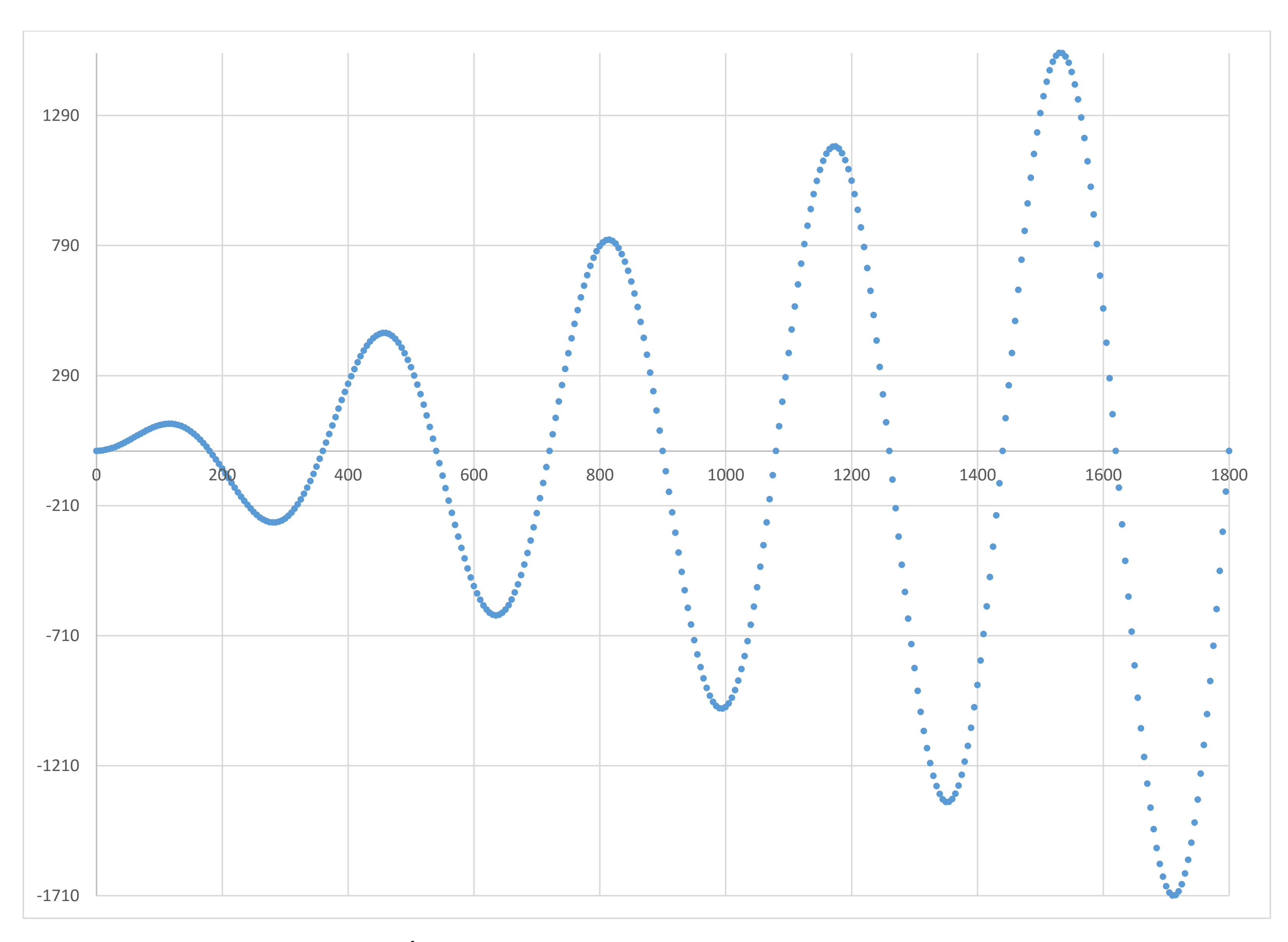
En los dos ejemplos anteriores, los valores de las entradas externas del perceptrón fueron 0 o 1. Pero no está limitado a eso, las entradas externas pueden tener valores entre 0 y 1 (incluyendo el 0 y el 1) por ejemplo: 0.7321, 0.21896, 0.9173418

El problema que se plantea es dado el comportamiento de un evento en el tiempo, ¿podrá la red neuronal deducir el patrón?

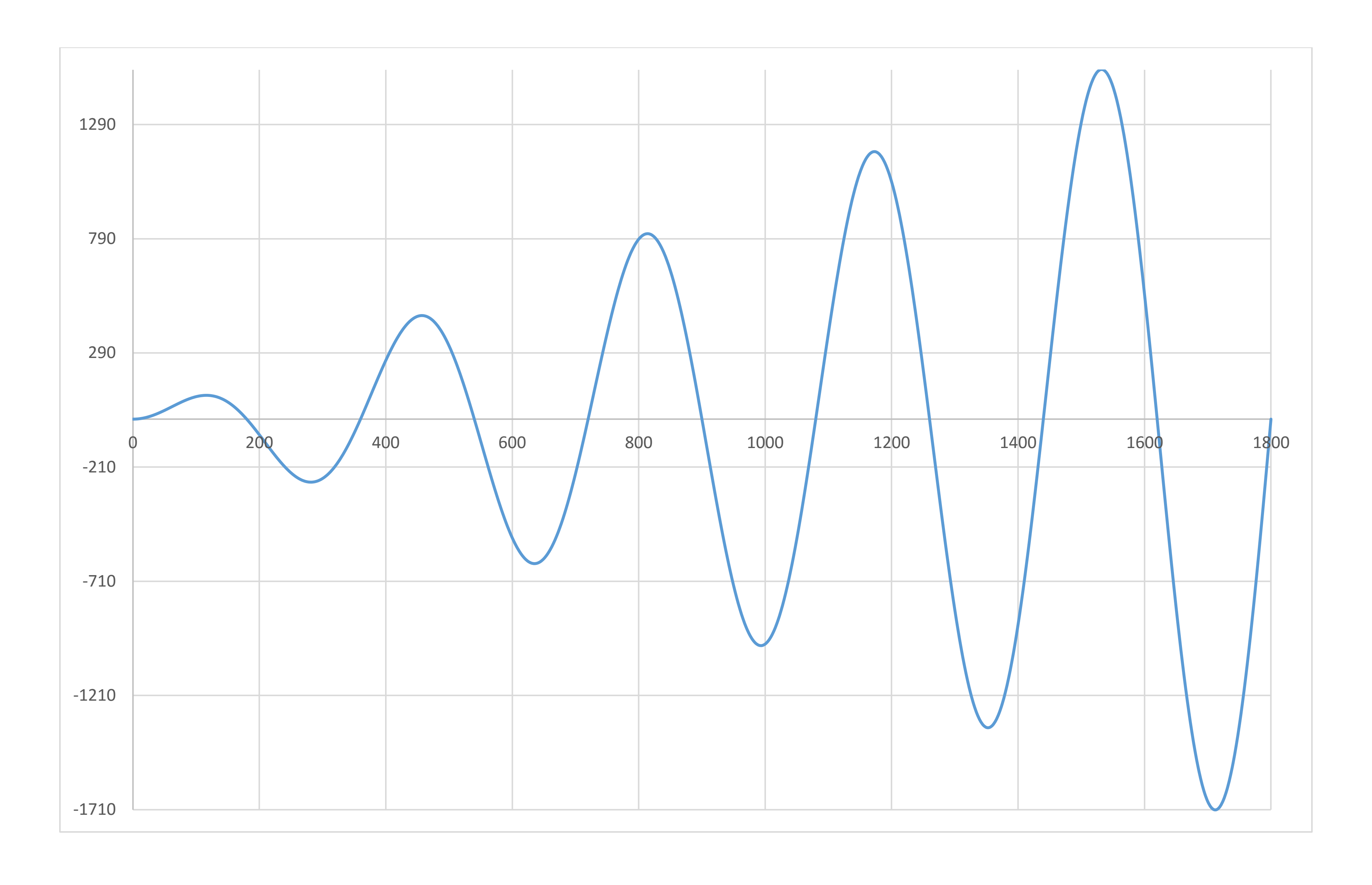
Ejemplo: Tenemos esta tabla

X	Y
0	0
5	0,43577871
10	1,73648178
15	3,88228568
20	6,84040287
25	10,5654565
30	15
35	20,0751753
40	25,7115044
45	31,8198052
50	38,3022222
55	45,0533624
60	51,9615242
65	58,9100062
70	65,7784835
75	72,444437
80	78,7846202
85	84,6765493
90	90
95	94,6384963
100	98,4807753

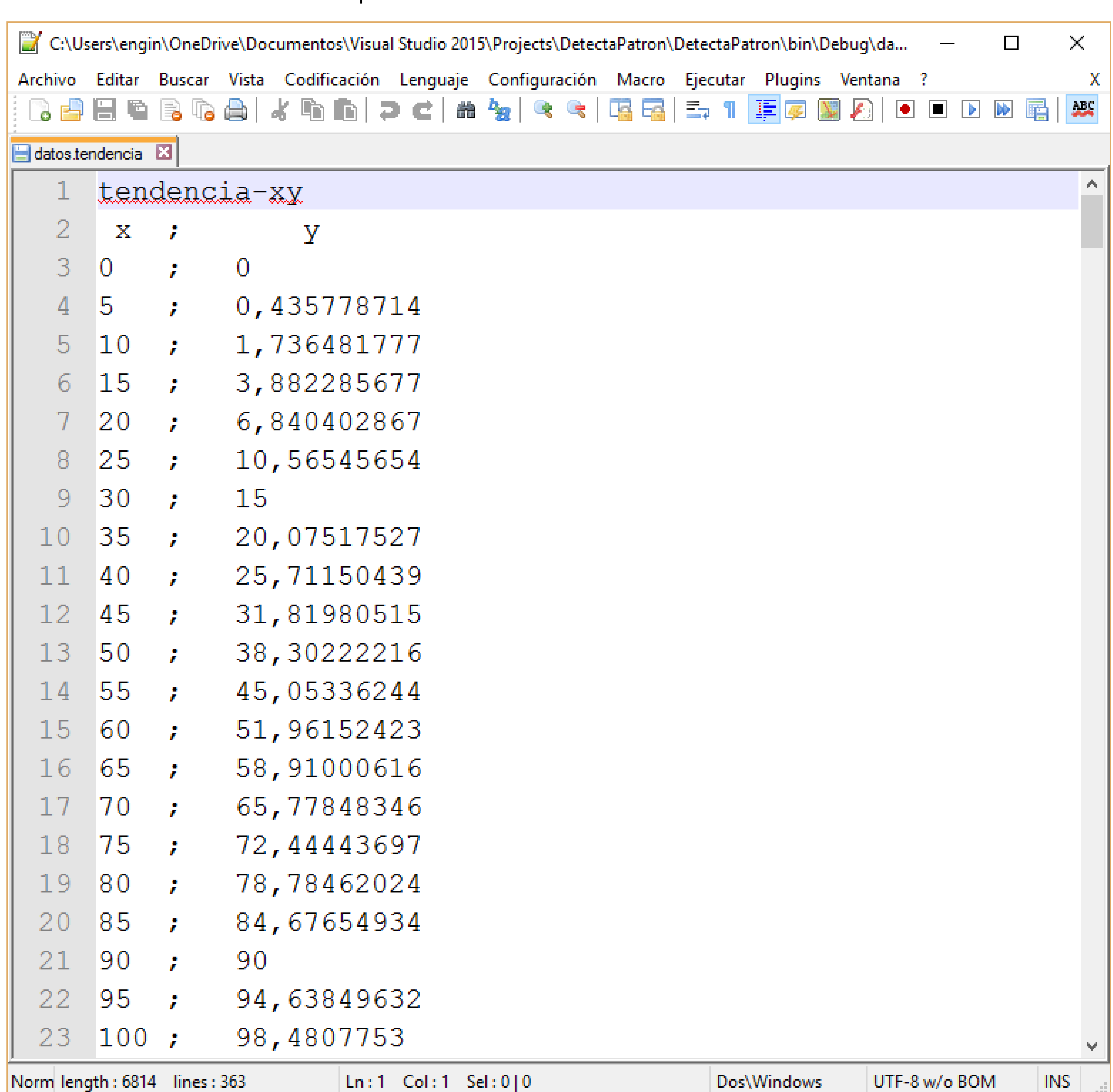
X es la variable independiente y Y es la variable dependiente, en otras palabras y=f(x). El problema es que no sabemos f( ), sólo tenemos los datos (por motivos prácticos se muestra la tabla con X llegando hasta 100, realmente llega hasta 1800). Esta sería la gráfica.



Uniendo los puntos, se obtendría



Los datos están en un archivo plano



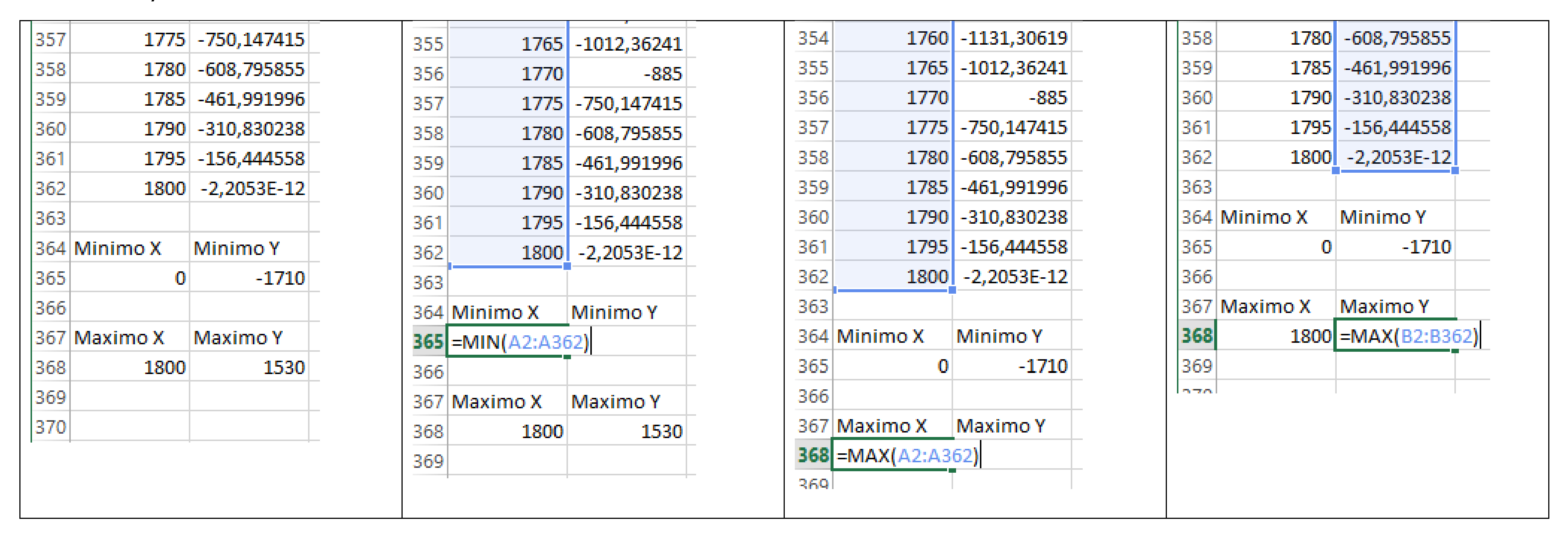
El primer paso es convertir los valores de X y de Y que se leen del archivo plano, en valores entre 0 y 1 (porque esa es la limitante del perceptrón multicapa). La razón de eso es que la función de activación del perceptrón multicapa $y = \frac{1}{1+e^{-x}}$ solo genera números entre 0 y 1.

Se hace entonces una normalización usando la siguiente fórmula

$$X_{normalizado} = \frac{X_{original} - MinimoX}{MaximoX - MinimoX}$$

$$Y_{normalizado} = \frac{Y_{original} - MinimoY}{MaximoY - MinimoY}$$

Como indica la fórmula, habría que recorrer todos los datos de X y Y para deducir el mayor valor de X, menor valor de X, mayor valor de Y y menor valor de Y.



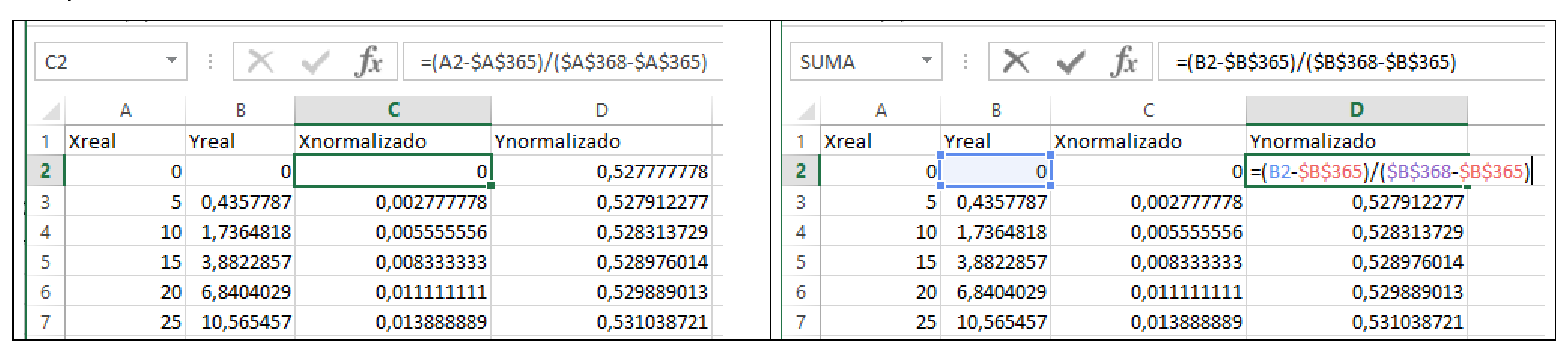
357	1775	-750,147415
358	1780	-608,795855
359	1785	-461,991996
360	1790	-310,830238
361	1795	-156,444558
362	1800	-2,2053E-12
363		
364	Minimo X	Minimo Y
365	0	-1710
366		
367	Maximo X	Maximo Y
368	1800	1530
369		
370		

355	1765	-1012,36241
356	1770	-885
357	1775	-750,147415
358	1780	-608,795855
359	1785	-461,991996
360	1790	-310,830238
361	1795	-156,444558
362	1800	-2,2053E-12
363		
364	Minimo X	Minimo Y
365	=MIN(A2:A362)	
366		
367	Maximo X	Maximo Y
368	1800	1530
369		

354	1760	-1131,30619
355	1765	-1012,36241
356	1770	-885
357	1775	-750,147415
358	1780	-608,795855
359	1785	-461,991996
360	1790	-310,830238
361	1795	-156,444558
362	1800	-2,2053E-12
363		
364	Minimo X	Minimo Y
365	0	-1710
366		
367	Maximo X	Maximo Y
368	=MAX(A2:A362)	
369		

358	1780	-608,795855
359	1785	-461,991996
360	1790	-310,830238
361	1795	-156,444558
362	1800	-2,2053E-12
363		
364	Minimo X	Minimo Y
365	0	-1710
366		
367	Maximo X	Maximo Y
368	1800	=MAX(B2:B362)
369		

Se aplica la fórmula



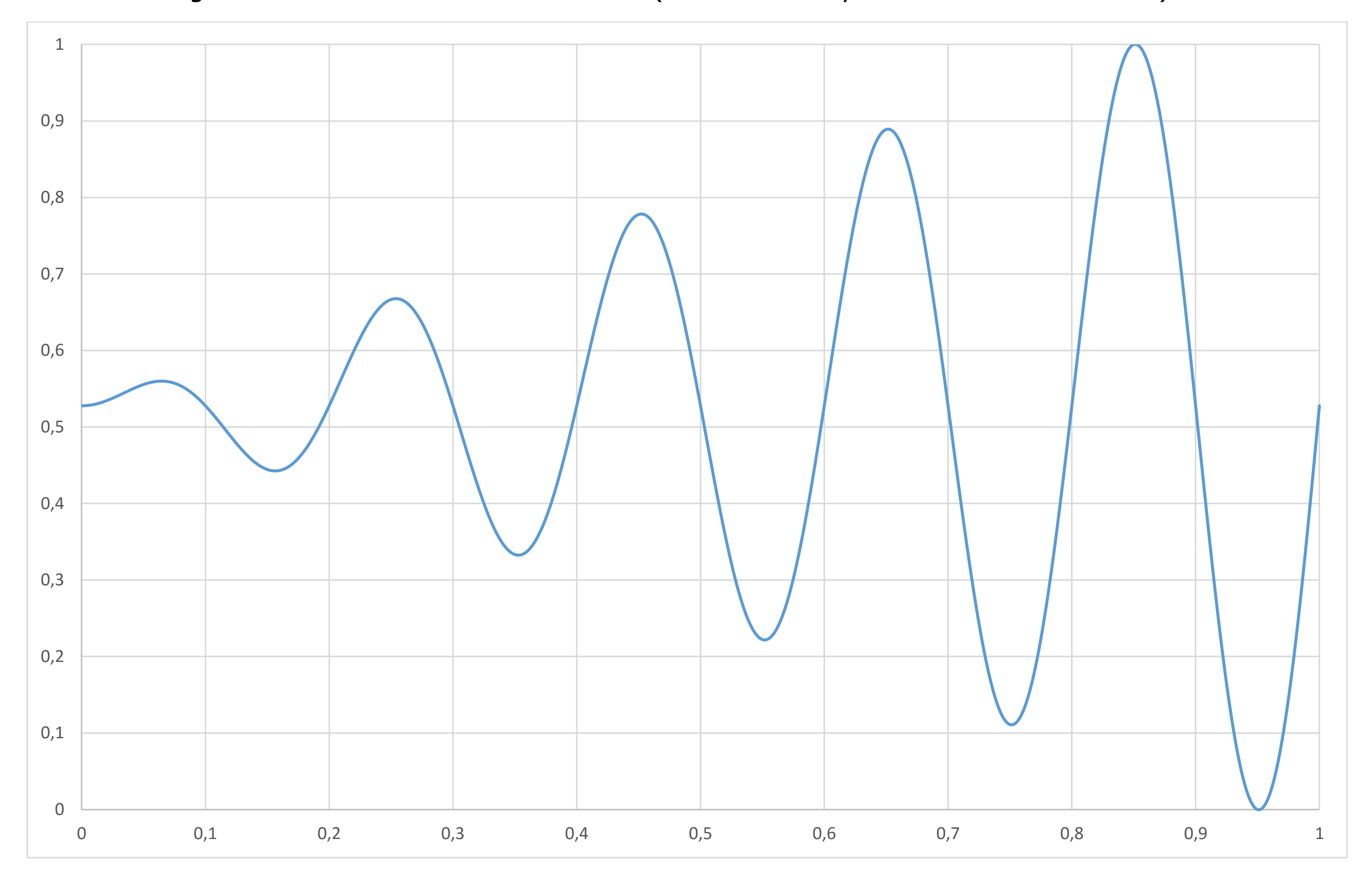
C2 =(A2-$A$365)/($A$368-$A$365)

	A	B	C	D
1	Xreal	Yreal	Xnormalizado	Ynormalizado
2	0	0	0	0,527777778
3	5	0,4357787	0,002777778	0,527912277
4	10	1,7364818	0,005555556	0,528313729
5	15	3,8822857	0,008333333	0,528976014
6	20	6,8404029	0,011111111	0,529889013
7	25	10,565457	0,013888889	0,531038721

SUMA =(B2-$B$365)/($B$368-$B$365)

	A	B	C	D
1	Xreal	Yreal	Xnormalizado	Ynormalizado
2	0	0	0	=(B2-$B$365)/($B$368-$B$365)
3	5	0,4357787	0,002777778	0,527912277
4	10	1,7364818	0,005555556	0,528313729
5	15	3,8822857	0,008333333	0,528976014
6	20	6,8404029	0,011111111	0,529889013
7	25	10,565457	0,013888889	0,531038721

Se realiza la gráfica con los valores normalizados (datos entre 0 y 1 tanto en X como en Y)



Con esos datos normalizados, se alimenta la red neuronal para entrenarla. Una vez termine el entrenamiento se procede a "desnormalizar" los resultados. A continuación el código completo:

```
using System;

namespace DetectaPatron {
    //La clase que implementa el perceptrón
    class Perceptron {
        private double[,,] W; //Los pesos serán arreglos multidimensionales. Así: W[capa, neurona inicial, neurona final]
        private double[,] U; //Los umbrales de cada neurona serán arreglos bidimensionales. Así: U[capa, neurona que produce la salida]
        double[,] A; //Las salidas de cada neurona serán arreglos bidimensionales. Así: A[capa, neurona que produce la salida]

        private double[,,] WN; //Los nuevos pesos serán arreglos multidimensionales. Así: W[capa, neurona inicial, neurona final]
        private double[,] UN; //Los nuevos umbrales de cada neurona serán arreglos bidimensionales. Así: U[capa, neurona que produce la salida]

        private int TotalCapas; //El total de capas que tendrá el perceptrón incluyendo la capa de entrada
        private int[] neuronasporcapa; //Cuantas neuronas habrá en cada capa
        private int TotalEntradas; //Total de entradas externas del perceptrón
        private int TotalSalidas; //Total salidas externas del perceptrón

        public Perceptron(int TotalEntradas, int TotalSalidas, int TotalCapas, int[] neuronasporcapa) {
            this.TotalEntradas = TotalEntradas;
            this.TotalSalidas = TotalSalidas;
            this.TotalCapas = TotalCapas;
            int maxNeuronas = 0; //Detecta el máximo número de neuronas por capa para dimensionar los arreglos
            this.neuronasporcapa = new int[TotalCapas + 1];
            for (int capa = 1; capa <= TotalCapas; capa++) {
                this.neuronasporcapa[capa] = neuronasporcapa[capa];
                if (neuronasporcapa[capa] > maxNeuronas) maxNeuronas = neuronasporcapa[capa];
            }

            //Dimensiona con el máximo valor
            W = new double[TotalCapas + 1, maxNeuronas + 1, maxNeuronas + 1];
            U = new double[TotalCapas + 1, maxNeuronas + 1];
            WN = new double[TotalCapas + 1, maxNeuronas + 1, maxNeuronas + 1];
            UN = new double[TotalCapas + 1, maxNeuronas + 1];
            A = new double[TotalCapas + 1, maxNeuronas + 1];

            //Da valores aleatorios a pesos y umbrales
            Random azar = new Random();

            for (int capa = 2; capa <= TotalCapas; capa++)
                for (int i = 1; i <= neuronasporcapa[capa]; i++)
                    U[capa, i] = azar.NextDouble();

            for (int capa = 1; capa < TotalCapas; capa++)
                for (int i = 1; i <= neuronasporcapa[capa]; i++)
                    for (int j = 1; j <= neuronasporcapa[capa + 1]; j++)
                        W[capa, i, j] = azar.NextDouble();
        }

        public void Procesa(double[] E) {
            //Entradas externas del perceptrón pasan a la salida de la primera capa
            for (int copia = 1; copia <= TotalEntradas; copia++) A[1, copia] = E[copia];

            //Proceso del perceptrón
            for (int capa = 2; capa <= TotalCapas; capa++)
                for (int neurona = 1; neurona <= neuronasporcapa[capa]; neurona++) {
                    A[capa, neurona] = 0;
                    for (int entra = 1; entra <= neuronasporcapa[capa - 1]; entra++)
                        A[capa, neurona] += A[capa - 1, entra] * W[capa - 1, entra, neurona];
                    A[capa, neurona] += U[capa, neurona];
                    A[capa, neurona] = 1 / (1 + Math.Exp(-A[capa, neurona]));
                }
        }

        // Muestra las entradas externas del perceptrón, las salidas esperadas y las salidas reales
        public void Muestra(double[] E, double[] S, double minimoX, double maximoX, double minimoY, double maximoY) {
            //Console.Write(E[1]*(maximoX- minimoX)+minimoX); Console.Write(" ===> ");
            //Console.Write(S[1]*(maximoY- minimoY)+minimoY); Console.Write(" <vs> ");
            Console.WriteLine(A[TotalCapas, 1]*(maximoY- minimoY)+minimoY); //Salidas reales del perceptrón
        }

        //El entrenamiento es ajustar los pesos y umbrales
        public void Entrena(double alpha, double[] E, double[] S) {
            //Ajusta pesos capa3 ==> capa4
            for (int j = 1; j <= neuronasporcapa[3]; j++)
                for (int i = 1; i <= neuronasporcapa[4]; i++) {
                    double Yi = A[4, i];
                    double dE3 = A[3, j] * (Yi - S[i]) * Yi * (1 - Yi);
                    WN[3, j, i] = W[3, j, i] - alpha * dE3; //Nuevo peso se guarda temporalmente
                }

            //Ajusta pesos capa2 ==> capa3
            for (int j = 1; j <= neuronasporcapa[2]; j++)
                for (int k = 1; k <= neuronasporcapa[3]; k++) {
                    double acum = 0;
                    for (int i = 1; i <= neuronasporcapa[4]; i++) {
                        double Yi = A[4, i];
                        acum += W[3, k, i] * (Yi - S[i]) * Yi * (1 - Yi);
                    }
                    double dE2 = A[2, j] * A[3, k] * (1 - A[3, k]) * acum;
                    WN[2, j, k] = W[2, j, k] - alpha * dE2; //Nuevo peso se guarda temporalmente
                }

            //Ajusta pesos capa1 ==> capa2
            for (int j = 1; j <= neuronasporcapa[1]; j++)
                for (int k = 1; k <= neuronasporcapa[2]; k++) {
                    double acumular = 0;
                    for (int p = 1; p <= neuronasporcapa[3]; p++) {
                        double acum = 0;
                        for (int i = 1; i <= neuronasporcapa[4]; i++) {
```

```
                        double Yi = A[4, i];
                        acum += W[3, p, i] * (Yi - S[i]) * Yi * (1 - Yi);
                    }
                    acumular += W[2, k, p] * A[3, p] * (1 - A[3, p]) * acum;
                }
                double dE1 = E[j] * A[2, k] * (1 - A[2, k]) * acumular;
                WN[1, j, k] = W[1, j, k] - alpha * dE1; //Nuevo peso se guarda temporalmente
            }

            //Ajusta umbrales de neuronas de la capa 4
            for (int i = 1; i <= neuronasporcapa[4]; i++) {
                double Yi = A[4, i];
                double dE4 = (Yi - S[i]) * Yi * (1 - Yi);
                UN[4, i] = U[4, i] - alpha * dE4; //Nuevo umbral se guarda temporalmente
            }

            //Ajusta umbrales de neuronas de la capa 3
            for (int k = 1; k <= neuronasporcapa[3]; k++) {
                double acum = 0;
                for (int i = 1; i <= neuronasporcapa[4]; i++) {
                    double Yi = A[4, i];
                    acum += W[3, k, i] * (Yi - S[i]) * Yi * (1 - Yi);
                }
                double dE3 = A[3, k] * (1 - A[3, k]) * acum;
                UN[3, k] = U[3, k] - alpha * dE3; //Nuevo umbral se guarda temporalmente
            }

            //Ajusta umbrales de neuronas de la capa 2
            for (int k = 1; k <= neuronasporcapa[2]; k++) {
                double acumular = 0;
                for (int p = 1; p <= neuronasporcapa[3]; p++) {
                    double acum = 0;
                    for (int i = 1; i <= neuronasporcapa[4]; i++) {
                        double Yi = A[4, i];
                        acum += W[3, p, i] * (Yi - S[i]) * Yi * (1 - Yi);
                    }
                    acumular += W[2, k, p] * A[3, p] * (1 - A[3, p]) * acum;
                }
                double dE2 = A[2, k] * (1 - A[2, k]) * acumular;
                UN[2, k] = U[2, k] - alpha * dE2; //Nuevo umbral se guarda temporalmente
            }

            //Copia los nuevos pesos y umbrales a los pesos y umbrales respectivos del perceptrón
            for (int capa = 2; capa <= TotalCapas; capa++)
                for (int i = 1; i <= neuronasporcapa[capa]; i++)
                    U[capa, i] = UN[capa, i];

            for (int capa = 1; capa < TotalCapas; capa++)
                for (int i = 1; i <= neuronasporcapa[capa]; i++)
                    for (int j = 1; j <= neuronasporcapa[capa + 1]; j++)
                        W[capa, i, j] = WN[capa, i, j];
        }
    }

    class Program {
        static void Main(string[] args) {
            int TotalEntradas = 1; //Número de entradas externas del perceptrón
            int TotalSalidas = 1; //Número de salidas externas del perceptrón
            int TotalCapas = 4; //Total capas que tendrá el perceptrón
            int[] neuronasporcapa = new int[TotalCapas + 1]; //Los índices iniciarán en 1 en esta implementación
            neuronasporcapa[1] = TotalEntradas; //Entradas externas del perceptrón
            neuronasporcapa[2] = 8; //Capa oculta con 8 neuronas
            neuronasporcapa[3] = 8; //Capa oculta con 8 neuronas
            neuronasporcapa[4] = TotalSalidas; //Capa de salida con 1 neurona

            Perceptron objP = new Perceptron(TotalEntradas, TotalSalidas, TotalCapas, neuronasporcapa);

            //Lee los datos de un archivo plano
            int MaximosRegistros = 2000;
            double[][] entrada = new double[MaximosRegistros + 1][];
            double[][] salidas = new double[MaximosRegistros + 1][];
            const string urlArchivo = "datos.tendencia";
            int ConjuntoEntradas = LeeDatosArchivo(urlArchivo, entrada, salidas);

            //Normaliza los valores entre 0 y 1 que es lo que requiere el perceptrón
            double minimoX = entrada[1][1], maximoX = entrada[1][1];
            double minimoY = salidas[1][1], maximoY = salidas[1][1];
            for (int cont = 1; cont <= ConjuntoEntradas; cont++) {
                if (entrada[cont][1] > maximoX) maximoX = entrada[cont][1];
                if (salidas[cont][1] > maximoY) maximoY = salidas[cont][1];
                if (entrada[cont][1] < minimoX) minimoX = entrada[cont][1];
                if (salidas[cont][1] < minimoY) minimoY = salidas[cont][1];
            }

            for (int cont = 1; cont <= ConjuntoEntradas; cont++) {
                entrada[cont][1] = (entrada[cont][1] - minimoX) / (maximoX - minimoX);
                salidas[cont][1] = (salidas[cont][1] - minimoY) / (maximoY - minimoY);
            }

            //Inicia el proceso de la red neuronal
            double alpha = 0.4; //Factor de aprendizaje
            for (int epoca = 1; epoca <= 64000; epoca++) {
                if (epoca % 4000 == 0) Console.WriteLine("Iteracion: " + epoca);
                //Importante: Se envía el primer conjunto de entradas-salidas, luego el segundo, tercero y cuarto
                //por cada ciclo de entrenamiento.
                for (int entra = 1; entra <= ConjuntoEntradas; entra++) {
                    objP.Procesa(entrada[entra]);
                    objP.Entrena(alpha, entrada[entra], salidas[entra]);
                }
```

```
            }

            //Muestra el resultado
            for (int entra = 1; entra <= ConjuntoEntradas; entra++) {
                objP.Procesa(entrada[entra]);
                objP.Muestra(entrada[entra], salidas[entra], minimoX, maximoX, minimoY, maximoY);
            }

            Console.ReadKey();
        }

        private static int LeeDatosArchivo(string urlArchivo, double[][] entrada, double[][] salida) {
            var archivo = new System.IO.StreamReader(urlArchivo);
            archivo.ReadLine(); //La línea de simple serie
            archivo.ReadLine(); //La línea de título de cada columna de datos
            string leelinea;

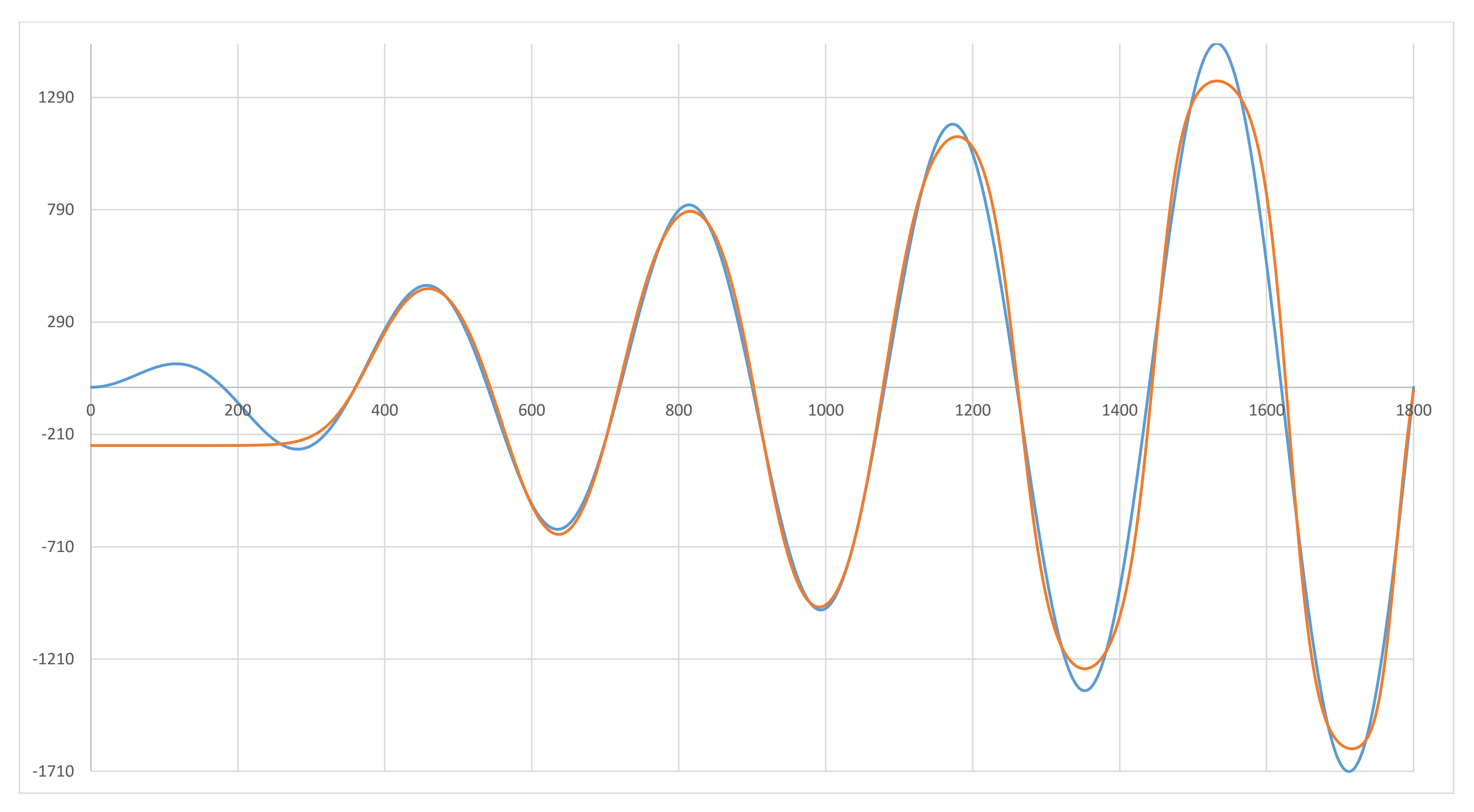
            int limValores = 0;
            while ((leelinea = archivo.ReadLine()) != null) {
                limValores++;
                double valX = TraerNumeroCadena(leelinea, ';', 1);
                double valY = TraerNumeroCadena(leelinea, ';', 2);
                entrada[limValores] = new double[] { 0, valX };
                salida[limValores] = new double[] { 0, valY };
            }
            archivo.Close();
            return limValores;
        }

        //Dada una cadena con separaciones por delimitador, trae determinado ítem
        private static double TraerNumeroCadena(string linea, char delimitador, int numeroToken) {
            string numero = "";
            int numTrae = 0;
            foreach (char t in linea) {
                if (t != delimitador)
                    numero = numero + t;
                else {
                    numTrae = numTrae + 1;
                    if (numTrae == numeroToken) {
                        numero = numero.Trim();
                        if (numero == "") return 0;
                        return Convert.ToDouble(numero);
                    }
                    numero = "";
                }
            }
            numero = numero.Trim();
            if (numero == "") return 0;
            return Convert.ToDouble(numero);
        }
    }
}
```

Ejemplo de ejecución

```
file:///C:/Users/engin/onedrive/documentos/visual studio 2015/Projects/DetectaPatron/DetectaPatron/bin/Debug/DetectaPatron.EXE
Iteracion: 4000
Iteracion: 8000
Iteracion: 12000
Iteracion: 16000
Iteracion: 20000
Iteracion: 24000
Iteracion: 28000
Iteracion: 32000
Iteracion: 36000
Iteracion: 40000
Iteracion: 44000
Iteracion: 48000
Iteracion: 52000
Iteracion: 56000
Iteracion: 60000
Iteracion: 64000
-259,557168481454
-259,556936534444
-259,556679694628
-259,55639491942
-259,556078722451
-259,555727097332
-259,555335426557
-259,554898372106
-259,554409743977
-259,553862341341
-259,553247760414
-259,552556161006
-259,551775982184
-259,550893594705
```

Llevando el resultado numérico al gráfico original, en azul los datos esperados, en naranja lo aprendido por la red neuronal.



Este proceso es lento, inclusive en un Intel Core i7. Puede acelerarse disminuyendo el número de neuronas en las capas ocultas a 5 (en ambas), mejorará la velocidad, pero bajará la precisión del ajuste en curva.

Bibliografía y webgrafía

Parte matemática

Redes Neuronales: Fácil y desde cero.

Autor: Javier García

https://www.youtube.com/watch?v=jaEIv_E29sk&list=PLAnA8FVrBl8AWkZmbswwWiF8a_52dQ3JQ